8086 ASSEMBLY LANGUAGE

8086アセンブリ言語

西村義孝 *yoshitaka NISHIMURA*

8086アセンブリ語

西村義孝

日本ソフトバンク

序

PC-9801が発表されてもう4年が過ぎようとしています。この間にPC-9801はPC-9801，PC-9801/ E / F，PC-9801M，PC-9801U，PC-9801VM / VF と着実な進歩を遂げてきました。またソフトウェアも他機種に類を見ないほど充実してきています。ワープロ，データベース，ゲーム等々，われわれが必要とするものは大半手に入る状態だといえます。また OS も CP/M86 に始まって MS-DOS V1.25，2.0，UCSD p-SYSTEM，PC-UXなど8ビット用 OS に毛のはえたようなものからミニコンの OS にせまるものまで自由に選択できます。OS の下で使えるソフトウェアも FORTRAN，COBOL，Pascal，LISP などの高級言語がお金さえ出せば自由になります。大変幸せな時代になったものです。それではわれわれはプログラムする必要がなくなったかというとそうでもありません。自分の目的に完全に合致したプログラムは，やはり自分で組まなければなりません。最初は BASIC で組んでみます。一応動くには動くがどうも速さに納得できません。次はコンパイラを使ってみます。若干速くなりますが，やはり納得できません。そしてどうしても機械語が必要になってきます。PC-9801の真の実力は，機械語プログラムでないと十分には発揮されないでしょう。目的のプログラムを全部機械語で組む必要はありません。機械語プログラムは「必要なとき，必要なだけ」使うべきです。そして「必要なとき」にはためらうことなく機械語が使えるようになってほしいものです。本書がその一助になれば幸いです。

この本は PC-9801 の CPU である8086の各命令に始まって，アセンブラの使い方，そして PC-9801 の売りの1つである GDC のプログラミングまでを解説しています。8086の入門書としても GDC の入門書としても存分に活用してほしいものです。

なお出版にあたっては日本ソフトバンク書籍部および Oh!PC 編集室の方々に大変お世話になりました。ここで感謝の意を表したいと思います。

1985年11月　　　　西村義孝

目　次

第1章　アセンブリ言語とは

第2章 8086の命令

第3章 ASM86の使い方

第4章　PC-9801のグラフィック

第1章
アセンブリ言語とは

1 はじめに

PC-9801が発売されてから4年がたちました。PC-9801用のソフトウェアもかなり出回り，アセンブリ言語によるゲームも多数見受けられます。「あんなゲームをなんとか自分の手で」と考えている方も多いでしょう。

ところが，いざアセンブリ言語を研究しようとしても，なかなかよい入門書がありません。8086の本はかなりあるのですが，大半が8ビットのアセンブラを使ったことのある人を対象としているので，「PC-9801が初めてのパソコン」である方や「BASICは分かるがアセンブリ言語は初めて」の方は困ってしまうのも事実でしょう。かといって，8ビットのアセンブリ言語から勉強するのは9801を現に持っている方にとっては二度手間としか思えません。

そこで本書では，アセンブリ言語はまったく初めての方にも理解できるように話を進めてゆきます。ただしBASICは十分理解しているものとします。BASICが分からずアセンブリ言語をやるのは，6，7年前ならともかく，いまでは主客転倒でしょう。

アセンブリ言語の必要性

以前アセンブリ言語入門の記事に対して，次のような投書がありました。

「16ビットでどうしてアセンブラがいるのか！」

おそらく「16ビットは十分速いので，C言語などの高速なコンパイラを使えばアセンブリ言語など必要ない！」というつもりでしょう。ところがそうはいかないのです。3DパッケージをためしにC言語で書いたところ，けっこう速いがゲームに使えるものではありませんでした。

確かに，C言語やPL/Mなどは高速ですが，コンパイラである以上，どうしても最適化しきれない部分[注1]が残ります。したがって，アセンブリ言語のプログラムのほうが高速になるわけです。人間の欲はきりがないもので，高速になれば処理させたいことはいくらでも増えてくるものです。ですから，アセンブリ言語はこれからも必要なくなることはないはずです。

アセンブリ言語を使うとき

ここで間違えてほしくないのは，「何でもかんでもアセンブリ言語で記述したほうがよい」といっているわけではありません。アセンブリ言語は高速な半面，開発効率は非常に悪いのです。同じプログラムを書くのに，BASICの何倍もの時間がかかります。BASICやコンパイラなどで，「どうしてもこのサブルーチンが遅い」というときに，そのサブルーチンだけをアセンブリ言語で書くといったやり方がよいと思います。

アセンブリ言語は「必要なとき，必要なだけ」使うべきです。間違っても，ウォーシミュレーションなど，処理速度が問題にならないものをすべてアセンブリ言語で書くようなことはしないでください。

アセンブリ言語は難しいか？

アセンブリ言語は難しいと敬遠されがちですが，そんなに難しいものではありません。「難しい」というよりも「面倒くさい」というべきでしょう。アセンブリ言語は，非常に細かい部分までプログラミングできるとも，細かく記述しなければならないともいえるでしょう。言語自体は中学生でも十分理解できるほど単純です。アセンブリ言語をものにできるかどうかは，根気があるかどうかにかかっています。

何が必要か

アセンブリ言語を使ってプログラムを開発するには，CP/M–86やMS–DOSなどのDOSにのっているアセンブラを使うのが理想的です。1つくらいDOSを買っておいても損はないでしょう。

PC–9801では，ディスクベース時にモニタのAコマンドでアセンブルすることができます。短いプログラムならば，これを利用するのもよいでしょう。ただ，機能的にはニーモニックアセンブラで低レベルですから，このアセンブラでゲームを作るのは考えものです。

ディスクがない場合は，OSはおろかモニタのAコマンドさえ使えません。したがって，テープベースのアセンブラを買ってくるか，ハンドアセンブルといって，人間がオペコード表を見ながら機械語に変換するしか方法はありませ

ん。本格的にアセンブリ言語を使いたい方にはディスクドライブは必要不可欠です。

本書では先にも述べたとおりBASICの分かっている方を対象に，基礎からDOSのアセンブラで自由にアセンブルできるまでを解説します。独力でアセンブリ言語を使えるためには，最低何を知っていなければならないか，それを説明したいと思います。

最後まで読み通して，アセンブリ言語を自分のものにしてください。

アセンブリ言語とは

コンピュータは0と1しか理解できないといわれていますが，マイクロコンピュータでも事情は同じです。

10110000

00000110

11100110

00110111

といった2進数のデータしかマイコンは理解できません。2進数では冗長になるので，マイコンでは2進数との対応がよくつき簡単な，16進表記をよく使います。

BASICのHEX$, &Hでなじみのある方も多いでしょう。いまのデータを16進で表せば，

B0H，06H，E6H，37H

と簡潔になります。BASICでは，16進数は

&HB006

などと**&H**をつけて表しましたが，アセンブリ言語では，

0B006H，0E637H

のように**H**をつけて表します（HはHexadecimal：16進数の略）。同じように2進数は，

10110000B，00000110B

とBをつけます（BはBinary：2進数の略）。

問題なのはマイコンの直接理解できる言葉が，

B0，06，E6，37

といった数値であることです。これを機械語（Machine Language）と呼びますが，この機械語を使って人間がプログラミングするのは非常に大変です。いまのB0，06，E6，37という機械語から，マイコンがどのような動作をするかを即座に理解できる人はほとんどいないでしょう。

これは，機械語が人間の言葉からかけ離れたものだからです。コンピュータの黎明期には，この機械語でプログラミングしていたわけですが，これでは開

発効率が悪すぎます。そこで考え出されたのが，**アセンブリ言語**です。アセンブリ言語は，B0，06，E6，37 といった機械語と1対1に対応した人間の言葉に近い言語です。

この B0，06，E6，37 をアセンブリ言語で書くと，

MOV AL，6
OUT 37H，AL

とかなり分かりやすい形になります。MOV は Move（転送）の意味で，OUT は BASIC の OUT 命令と同じです。

アセンブリ言語はいまのような MOV，OUT のほか HLT（Halt ：停止），ADD (ADDITION：加算)，SUB (SUBTRACTION：減算)，MUL (MULTIPLY：乗算)，DIV (DIVISION：除算）といった，英語の略号を使ってプログラミングしますから，機械語よりはるかに分かりやすいのです。本書でも，もちろんアセンブリ言語によるプログラミングを解説していきます。

アセンブラとは

いま，アセンブリ言語で

MOV AL，6
OUT 37H，AL

というプログラムを考えたとします。これをマイコンに実行させなければならないのですが，いかにアセンブリ言語が機械語に1対1に対応しているといっても，

MOV AL，6

などをマイコンは直接理解することはできません。マイコンはあくまで機械語しか分からないのです。そこで，アセンブリ言語から機械語に変換するプログラムが必要になるわけです。それを**アセンブラ**と呼びます。

アセンブラはつまり，

MOV AL，6
OUT 37H，A L } → B0 06 E6 37

という変換作業を行います。この変換は，**オペコード(OP-CODE)** 表という，マイコンの動作と機械語の対応表を参照しながら行われます。

この変換作業をアセンブルと呼びますが，もちろん人間の手で**オペコード**表

を見ながら変換することもできます。これを**ハンドアセンブル**と呼びますが，いまのように短いプログラムならともかく，長いプログラムでは変換するときに間違う可能性が高く，また変換に膨大な時間がかかるのでお勧めできません。

逆アセンブラという言葉を聞いたことがある方もいらっしゃるでしょう。逆アセンブラとは，機械語からアセンブリ言語に変換するものです。つまりアセンブラの逆の作業をします。

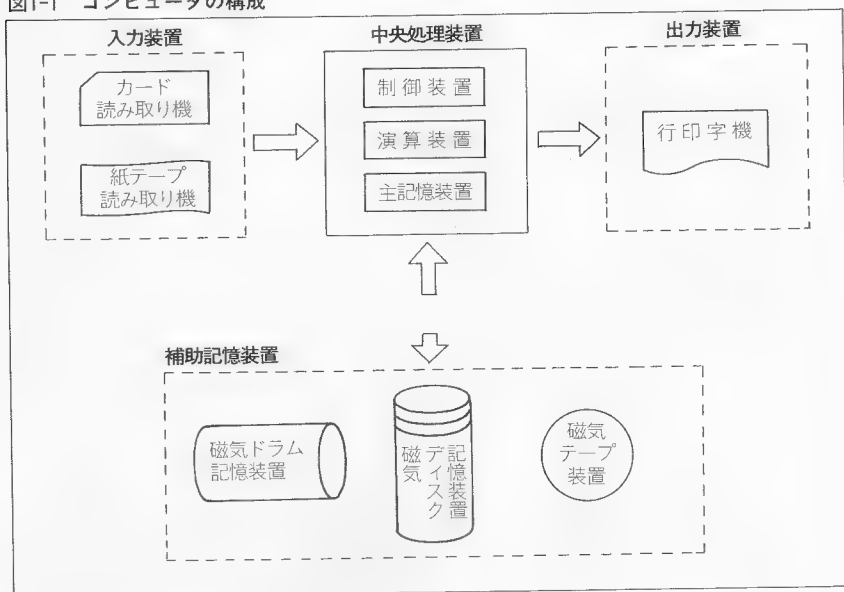
たとえば他人の作ったゲームなどを解析するのには，逆アセンブラを使います。ゲームはアセンブリ言語ではなく，機械語の形で売られているので，そのままでは分かりにくいのですが，逆アセンブラで逆アセンブルすれば，アセンブリ言語になりますから，かなり理解しやすくなります。PC-9801のモニタにも，ディスクBASICで使える逆アセンブル機能（Lコマンド）がついています。

逆アセンブラの機能を強力にしたものに，ソースジェネレータ[注2]というのもあります。

3 CPUについて

CPUとは中央処理装置（Central Processing Unit）のことで，大型コンピュータでは入出力装置，補助記憶装置以外のコンピュータの中心部をいいます（図1-1）。パソコンでも，本体部のことをCPUと呼ぶこともありますが，普通は本体内にあるマイクロプロセッサと呼ばれるLSIのことをCPUといいます。機械語プログラムは，このCPUが理解し実行するのです。

PC-9801には8086というCPUが使われています。8086はインテル社の開発した16ビット[注3] CPUです。インテル社は4004という世界初の4ビットCPUを開発して以来，8008, 8080, 8085などの8ビットCPU，8086, 80186などの16ビットCPUを次々とリリースしています。インテル社の8080は，8ビットCPUとしてはとても大きなシェアを誇っていました。現在の8ビット市場ではザイログ社のZ80が主流ですが，Z80も8080の完全上位互換性を保って作られたCPUです。

図1-1　コンピュータの構成

その後，インテル社は8085を開発しました。8085は8080の機械語を変更なしに実行でき，シリアルポートをもったCPUです。8085は主に制御の分野で使われています。

8086はこうした80系CPUの繁栄を踏まえて開発された16ビットCPUです。ですから，8086は8080や8085とアセンブリ言語のレベルで上位互換性があります(8085のシリアル関係の命令はもちろん互換性がありません。8086にはシリアルチャンネルはついていませんから)。事実，8080のアセンブリ言語プログラムを8086のプログラムに変換するソフトウェア（いわゆるトランスレータ）も売られています。

このため，8086用ソフトウェアは非常に多く，言語プロセッサでは8ビット用にリリースされているものは大半手に入る状態です。

ハードウェアに関しても8086は8080の周辺チップがかなり使えるため，16ビットCPUでは8086(8088)[注4]が最も普及しています。

16ビットCPUは，ほかにモトローラのMC68000，ザイログのZ8000などがあります。性能だけを比べると，8086よりも68000のほうがかなりまさっていますが，CPUは性能だけで普及するものではなく，ソフトウェア，ハードウェアをトータルに考えていかなくてはなりません。ただし，CPUの性能だけが問題になるスーパーパーソナルの分野では，68系CPUが主流となっていて，8086はほとんど顔を出しません。

8086は16ビットのローエンドといえます。

いまなぜ16ビットCPUなのか

〝8〟ビットCPUや〝16〟ビットCPUと説明なしに使ってきましたが，この数字はCPUが一度に処理できるデータの大きさを表しています。

8ビットだと0〜255の数値しか表現できません。すると8ビットCPUでは，

300＋1

を2回に分けて計算しなくてはなりません。つまり，300を（256＋44）と考えて，

（256＋44）＋1＝256＋45
　　　　　　　　＝301

と計算していくのです。このため処理時間がかかります。ところが16ビットでは，0〜65535の数値が使えますから，計算は一度ですみます。たいていの計算は0〜65535の間ですみますから，16ビットCPUのアセンブリ言語によるプログラムは，かなり組みやすいものとなります。このように16ビットCPUでは扱うデータが8ビットよりも大きくなると，8ビットCPUよりがぜん速くなります。1つの命令を処理するスピードも16ビットCPUでは速くなっていますから，同一プログラムでもかなりの速度向上が見られます。

パーソナルコンピュータや制御分野では8ビットCPUでも十分な場合が多いので，これからも8ビットCPUは使われていくでしょうが，パーソナルなユースでも計算速度が要求されることもあります。たとえば，三次元処理などです。

また，リアルタイム[注5]な応答が必要な場合，8ビットCPUでは力不足の場合が多いものです。そのため，パーソナルコンピュータでも16ビットCPUがかなり普及してきています。16ビットCPUで8ビットデータを扱うことも，もちろん可能です。特に8086ではストリング命令という8ビットデータを効率よく処理する命令群をもっているので，8ビットCPUより完全優位に立っています。

8086のレジスタ

レジスタとは，CPUが演算などをするために一時的に数値を記憶しておく

場所です。アセンブリ言語ではBASICの変数とほぼ同じような扱いを受けます。8086には，表1-1のようなレジスタがあります。8ビットCPUではこのレジスタの幅が8ビットの場合が多いのですが，16ビットCPUではほとんど

表1-1　レジスタ

16ビットの幅があります。先ほどの,

MOV **AL**, 6

OUT 37H, **AL**

の太字の AL がレジスタです。

8086は8080の上位互換性がありますから，レジスタも8080のものと対応させることもできます (表1-2)。

このため，8080のアセンブリ言語が使いこなせる人はすぐにでも8086を使えるようになるのですが，このコンパチビリティを追求したあまり，各レジスタごとに役割がほぼ決まっており，「この命令にはこのレジスタを使う」といった制約もあります。これを命令の非直交性と呼びます。

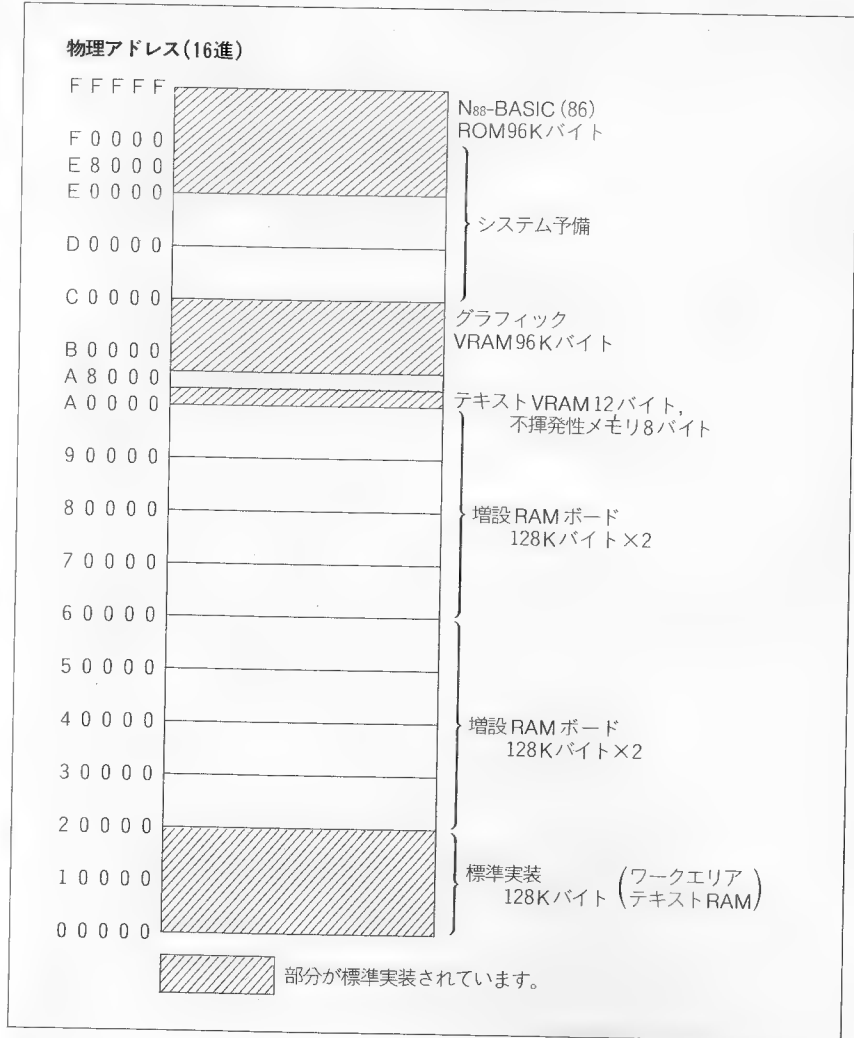
レジスタの使い方はこれから解説していきますが，8086にはこれだけのレジスタがあることを眺めておいてください。

表1-2　8086と8080の対応

8086	8080
AX	A
BX	H　L
CX	B　C
DX	D　E
F	PSW
SP	SP
IP	PC

メモリ空間

8086メモリ空間は1Mバイトです。メモリ空間とはBASICでいえばPEEK文，POKE文で読み出し，書き込みができる範囲といえるでしょう。メモリ

図1-2　メモリマップ

は１バイトを単位として，それが１M個あると考えてください。

８ビットCPUではメモリ空間は64Kバイトしかなく，VRAM[注6]を入れるエリアがなくなってしまい，バンク切り替え[注7]などを使っていましたが，8086では１Mバイトもありますから，VRAMもメモリ空間上に入ってしまいます（図1-2)。

セグメント 5

8086で難しいのはセグメントの概念でしょう。セグメントは8ビットCPUにはなかった考え方です。そのため，8ビットCPUのアセンブラを使ったことのある人もセグメントでつまずき，8086を自由に使えないことが多いようです。セグメント自体はそんなに難しくないのですが，実際にプログラミングしてみないと，なかなか身につかないものです。

8086のメモリ空間は1Mバイトあります。

$1\,M = 2^{20}$

ですから，1Mバイトを自由に指すためには，20ビットのレジスタが必要なのです。

ところが，8086には16ビットのレジスタしかありません。これでは

$2^{16} = 64K$（バイト）

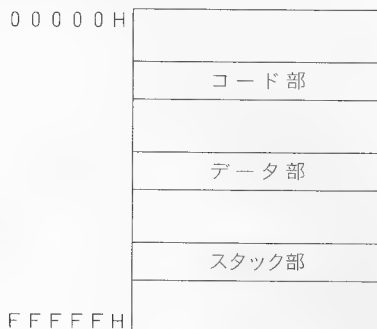
しか指し示すことができないはずです。

機械語プログラムは，普通

- コードの部分（プログラム本体）
- データの部分（変数，配列など）
- スタック[注8]の部分

の3つに分けられます（図1-3）。8086ではこの各部分を64Kバイト以内[注9]で作るのが普通です。64Kバイトもあれば，大半のプログラムは作成可能です。

図1-3　レジスタとそのビット長

アドレス	
00000H	
	コード部
	データ部
	スタック部
FFFFFH	

コード部，データ部，スタック部を各64Kバイト以内に作れば，その64Kバイト内であればどこでも16ビットレジスタで指し示すことができます。

このような64Kバイトの領域を，**セグメント**と呼びます。そして，いまのような

- コードの部分をコードセグメント
- データの部分をデータセグメント
- スタックの部分をスタックセグメント

と呼びます。8086はこのほかに，エクストラセグメント[注10]があります。

8086では1Mバイトのメモリ空間のどこにでも――16バイト単位で――セグメントを置くことができます。この16バイト単位のエリアをパラグラフと呼びます。0～15までを第0パラグラフ，16～31までを第1パラグラフ，……と呼びます。

16バイト単位で任意にセグメントを配置できるといいましたが，コード，データ，スタック，エクストラの各セグメントがどのパラグラフから始まっているかを8086に知らせなければなりません。これを設定するのが，**セグメントレジスタ**です。セグメントレジスタには図1-4のように

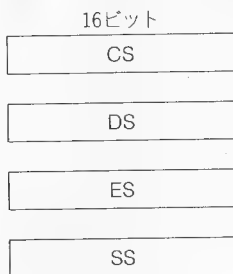
- CS（コードセグメントレジスタ）
- DS（データセグメントレジスタ）
- ES（エクストラセグメントレジスタ）
- SS（スタックセグメントレジスタ）

の4種類があります。

たとえば，データセグメントがアドレスの12340Hから始まっているとすれ

図1-4　セグメントレジスタ

16ビット

CS
DS
ES
SS

ば，12340Hはパラグラフ番号が1234H[注11]ですから，DSレジスタには1234Hをセットします。

このようにCS，DS，ES，SSには各セグメントの置かれている**パラグラフ番号**をセットするのです。パラグラフ番号は**セグメントアドレス**，または単にセグメントと呼ばれることもあります。

8ビットCPUは物理アドレスで示すことが多かったのですが，8086では物理アドレスのほかに

　　セグメントとオフセット

でアドレスを示すことがあります。**オフセット**とは，セグメント先頭からの距離のことです。

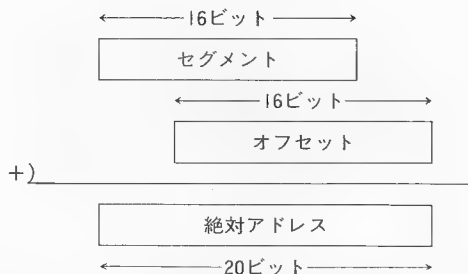
たとえば，物理アドレスの12345Hをセグメントとオフセットで表すと，

　　セグメント　1234H
　　オフセット　　　5H

と表せますし，

　　セグメント　1230H
　　オフセット　　45H

とも表せます。

要するに，

としてアドレスを示すのです。PC-9801ではグラフィックBIOSのワークエリアは，物理アドレスのC404Hからですが，通常は

　　セグメント　　60H
　　オフセット　640H～

と表します。つまり，

また，図1-5のメモリマップを見ると分かりますが，グラフィックVRAMは物理アドレスで，

青　A8000H～AFFFFH

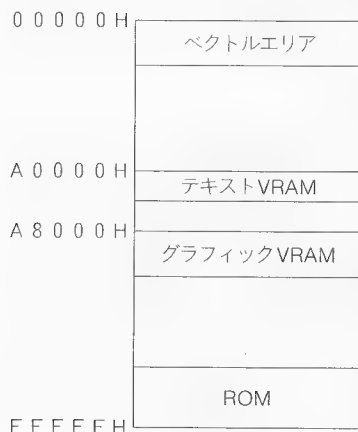
赤　B0000H～B7FFFH

緑　B8000H～BFFFFH

ですが，これも，

	セグメント	オフセット
青	A800H	0000H～7FFFH
赤	B000H	0000H～7FFFH
緑	B800H	0000H～7FFFH

となります。

図1-5　メモリマップ

BASICには

DEF SEG文

があります。これが自由に使える方はセグメントとオフセットがよく理解できているといえます。

たとえば，

「BASICでグラフィックVRAMをクリアするプログラム」

はどうなるでしょうか。

セグメントとオフセットが分かっている方なら，簡単に次のようなプログラムが書けるでしょう。

```
1000  FOR SEGMENT=&HA800  TO &HB800 STEP &H800
1010      DEF SEG=SEGMENT
1020      FOR OFFSET=0 TO &H7FFF
1030        POKE OFFSET, 0
1040      NEXT OFFSET
1050  NEXT SEGMENT
```

また，グラフィックVRAMをすべてディスクにセーブするプログラムは，

```
1000  INPUT "FILE NAME", F$
1010  FOR I=1 TO 3
1020    DEF SEG=&HA000+&H800*1
1030    BSAVE F$+STR$(I), 0, &H7FFF
1040  NEXT I
```

となります。このDEF SEG文が自由に使えないとセグメントとオフセットは理解できているとはいえません。

セグメントの概念はなかなか身につかないものです。セグメントとかオフセットがまったく初めての方には少し難しかったかもしれません。

DEF SEG文は自由に使えるようにしておいてください。

6 プログラム例

ごく簡単なプログラムを作成し，実行するまでを解説します。

「セグメント　B000H

オフセット　0000H

をFFHにするプログラムを書け」

BASICでは，

```
DEF SEG=&HB000
POKE 0, &HFF
```

となります。これをアセンブリ言語で書くと，

```
MOV AX, 0B000H ──────①
MOV DS, AX ──────────②
MOV AL, 0FFH ────────③
MOV DS:[0000H], AL ──④
```

となります。解説していきましょう。

①の

```
MOV AX, 0B000H
```

はAXレジスタに定数 0B000H をセットします。これはBASICで表すと，

```
AX=&HB000
```

となるでしょう。

MOV命令はこのように転送を行います。

前にも述べましたが，アセンブリ言語では，レジスタをBASICの変数のように使います。

②の

```
MOV DS, AX
```

はDSレジスタにAXレジスタの内容をセットします。BASICでは，

```
DS=AX
```

となるでしょう。この2つ

```
MOV AX, 0B000H
MOV DS, AX
```

により，DSレジスタには 0B000H がセットされました。それならば，

```
MOV     DS, 0B000H
```

とすればよいようなものです。ところが8086では，

「セグメントレジスタに定数を直接ロードできない」

のです。そのため，ほかの汎用レジスタ AX, BX, CX, DX, BP, SI, DI のどれかを使って間接的にロードします。このことは，8086のアセンブリ言語を使うときに，最初につまずく点です。「セグメントレジスタには直接，定数がロードできない」などというと，では「MOV は何と何の間の転送が可能なのか」という疑問がわいてきます。MOV 命令に限らず，ほかのすべての命令にもこのような制約がつきまといます。「MUL は定数乗算に使えない」とか「XLAT は BX と AL しか使えない」と各命令ごとに決まっています。それらをすべて覚えるのは大変です。

この決まりはすべてオペコード表に載っています。ですから，すべてを覚える必要はありません。そして，使っているうちに大半は覚えてしまいます。

話をプログラムに戻しましょう。

```
MOV     AX, 0B000H ────────①
MOV     DS, AX ────────────②
```

で

DS＝0B000H

となったわけです。これでデータセグメントは 0B000H からに指定できました。

③の

```
MOV     AL, 0FFH
```

は AL レジスタに 0FFH をセットします。

BASIC では

AL＝&HFF

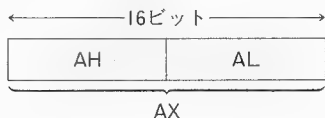
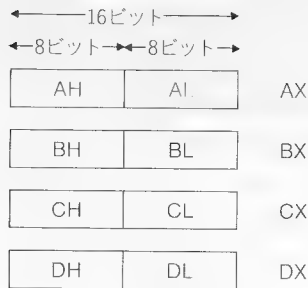
となるでしょう。AL レジスタは，AX レジスタの下位８ビットです。つまり

のようになっています。このように，上位下位に分けて使えるレジスタは図

図1-6　上位，下位に分けて使えるレジスタ

1-6の4つです。機械語プログラムで扱うデータは16ビットが多いのですが，文字データなどでは8ビットのデータも多く，このようにバイトデータを使いやすくするために，レジスタを上下に分割して使えるようになっています。ほかのレジスタ

　　SI，DI，BP，DS，ES，SS，IP

などは分割して使うことはできません。常に16ビット長のレジスタとして使います。

　　MOV　AL，0FFH

では，データがFFHで8ビットですから，ALレジスタを使いました。

　次に④の

　　MOV　DS:[0000H]，AL

では，ALレジスタの内容をデータセグメントのオフセット 0000H にセットします。いま，データセグメントレジスタDSはB000Hとなっていますから，めでたくセグメントB000H，オフセット0000HにFFHがセットされました。

いま，

　　MOV　DS:[0000H]，AL

と表記しましたが，このDS:は**セグメントオーバーライドプレフィックス**と呼ばれ，

　　CS:　DS:　ES:　SS:

があります。データはデータセグメントに含まれることが大半ですから，DS:に限り省略できます。[注12]

　だから，④の

MOV　DS: [0000H], AL

は,

MOV　[0000H], AL

と DS: を書かなくてもよいのです。

では

```
MOV    AX, 0B000H
MOV    DS, AX
MOV    AL, 0FFH
MOV    [0000H], AL
```

をアセンブルしてみましょう。

結果は,

```
     機械語                 アセンブリ言語
B8   00   B0        MOV    AX, 0B000H
8E   D8             MOV    DS, AX
B0   FF             MOV    AL, 0FFH
A2   00   00        MOV    [0000H], AL
```

となります。

7 アセンブル

MS–DOS, CP/M–86, モニタによる簡単なアセンブル方法を示します。使用したアセンブラは

MS–DOS ……… MACRO86
CP/M–86 ……… ASM86
モニタ………… Aコマンド

です。ASM86については2章で徹底的に解説しますので，ここでは使い方の実際を理解してください。

なお，実行は次項のとおりモニタで打ち込んだあと行ってください。

MS-DOS

まず，EDLINかほかのテキストエディタで，リスト1–1のソースリストを打ち込みます。

リスト1-1

```
TEST    SEGMENT                  TESTというSEGMENTの開始
        ASSUME  CS:TEST          TESTセグメントはコードセグメントであることを指定
        MOV     AX,0B000H      ┐
        MOV     DS,AX          │本文のとおり
        MOV     AL,0FFH        ┘
        MOV     DS:[0000H],AL    MACRO86では，ここのDS：は省略できない
TEST    ENDS                     TESTの終了
        END                      ソースの終了
```

ソースを打ち込んだら，それをアセンブラにかけます。ソースファイル名を

PC. ASM

(エクステンションはASM)

とすれば，

MASM PC, NULL, PC, NULL ⏎

でリスト1–2のファイルができ上がります。

リスト1-2

```
The Microsoft MACRO Assembler              11-07-85      PAGE     1-1


0000                           TEST     SEGMENT
                                        ASSUME  CS:TEST
0000  B8 B000                           MOV     AX,0B000H
0003  8E D8                             MOV     DS,AX
0005  B0 FF                             MOV     AL,0FFH
0007  A2 0000                           MOV     DS:[0000H],AL
000A                           TEST     ENDS
                                        END
 The Microsoft MACRO Assembler              11-07-85      PAGE    Symbols
                                -1


Segments and groups:

                N a m e                 Size    align   combine class

TEST . . . . . . . . . . . . . .        000A    PARA    NONE

Warning Severe
Errors  Errors
0       0
```

CP/M-86

EDやほかのテキストエディタでリスト1-3のソースプログラムを打ち込みます。

リスト1-3

```
MOV     AX,0B000H
MOV     DS,AX
MOV     AL,0FFH
MOV     .0000H,AL       ASM86では[0000H]の代わりに .0000Hを使う
END                     ソースの終了
```

このファイル名を

PC. A86

(エクステンションはA86)

とすれば,

ASM86 PC $HZ SZ ⏎

でリスト1-4のフィイル,

PC.LST

ができ上がります。

リスト1-4

```
CP/M ASM86 1.1  SOURCE: LIST3.A86
                                      PAGE    1

 0000 B800B0                     MOV      AX,0B000H
 0003 8ED8                       MOV      DS,AX
 0005 B0FF                       MOV      AL,0FFH
 0007 A20000                     MOV      .0000H,AL
                                 END

END OF ASSEMBLY.  NUMBER OF ERRORS:    0.  USE FACTOR:  0%
```

モニタ

MON ⏎

でモニタモードにし,

C1800 ⏎

A0 ⏎

のあと①～④を打ち込みます。そして,

L0 ⏎

とすればリスト1-5が表示されます。

リスト1-5

```
0000 B800B0          MOV      AX,B000
0003 8ED8            MOV      DS,AX
0005 B0FF            MOV      AL,FF
0007 A20000          MOV      [0000],AL
```

実　行　8

では，いまの機械語を実行しましょう。

8086にプログラムを実行させるには，メモリ上に機械語をセットしなくてはなりません。機械語のゲームを打ち込んだことのある人は分かるでしょうが，通常**モニタ**を使います。モニタを使えば，メモリ空間上のどこでも自由に機械語をセットできます。しかし，メモリ空間の一部はROMで書き込むことができませんし，RAMでもBASICインタプリタがワークエリアとして使っていたり，むやみに置くわけにはいきません。ここでは，

セグメント　1800H

オフセット　0000H

からデータをセットします。モニタの使い方は

MON　⏎

でモニタモードになり，

C1800　⏎

でこれから参照するアドレスがセグメント1800Hに含まれることを指定します。次に

E0000　⏎

として，機械語入力モードにします。あとは，さっきの機械語データ

B8　00　B0　8E　D8

B0　FF　A2　00　00

を打ち込みます。これで，このプログラムは，

セグメント　1800H

オフセット　0000H～0009H

にセットされました。

いよいよ実行です。実行にはGコマンドを使います。

G0000,　000A　⏎

で0000Hから実行させ，000AHで停止させます。

このプログラムでは，VRAMにデータを直接書き込んでいますから，画面の左上に赤い横線が出ればOKです。

アドレシングモード

オペコード表の読み方を説明する前に，どうしても知っておかなければならないことがあります。1つは**アドレシングモード**です。

INC　　AX

の INC は INCREMENT の略で，

「AX レジスタに 1 を加える」

という命令です。命令の多くはこのように「何を」，「どうする」という部分に分かれています。この「何を」と指し示すことを**アドレシング**と呼び，その指し示す方法を

「アドレシングモード」

と呼びます。いまの

INC　　AX

では，「AX レジスタに 1 を加える」というふうに，レジスタでアドレシングしているのです。また，

MOV　BX, AX ―――――――①

MOV [BX], AX ―――――――②

で，

①は「BX レジスタに AX レジスタの内容を転送する」

②は「BX レジスタの内容の指すメモリにAXレジスタの内容を転送する」

命令です。このように「何を」に当たる部分の指定方法（アドレシングモード）にはいくつかの種類があります。アドレシングモードは

- プログラム・メモリ・アドレシングモード
- データ・メモリ・アドレシングモード

の2つに分かれています。プログラム・メモリ・アドレシングモードは8086がプログラムを実行するときに関係するモードですが，これはジャンプ命令やコール命令と関係があります。

データ・メモリ・アドレシングモードは，8086がデータを扱うときのアドレシングモードです。

データ・メモリ・アドレシングモードは，いくつかの分け方がありますが，

ここでは次の７つに分類して説明しましょう。

- イミディエイト
- レジスタ
- ダイレクト
- レジスタ間接
- シングルインデックスト
- ダブルインデックスト
- スタック

それぞれについて説明しましょう。

イミディエイト・アドレシング

```
MOV    AX, 1234H ――――――③
```

というのがイミディエイト・アドレシングです。③は AX レジスタに数値 1234H を転送する命令です。この③の機械語は，あとで説明するように

```
B8    34    12
```

となりますが，34, 12と順番は逆になっているものの，1234H が機械語として表れています。イミディエイトは命令コード（この場合は B8）の直後のデータを数値として扱うアドレシングモードだといえます。数値データが上下逆になるのは 8080，Z80など80系では有名なことで，8080の16ビット版である8086でもこのように逆になります。6800，6809，68000 など68系の CPU の機械語ではこのようなことはありません。

レジスタ・アドレシング

```
MOV    AX, BX
```

これは，AX レジスタに BX レジスタの内容を転送する命令です。つまりレジスタ・アドレシングでは直後のレジスタの内容を指定します。

ダイレクト・アドレシング

```
MOV    AX, [1234H] ――――――④
```

これは，AX レジスタにアドレス 1234H の内容を転送します。80系ですから

AL レジスタに 1234H の内容が

AH レジスタに 1235H の内容が

それぞれ転送されます。④の機械語はあとで分かるように,

A1　34　12

となります。例によって順番は逆になっていますが, 1234H が機械語の中に表れています。ダイレクト・アドレシングとは, 命令コード（この場合は A1）の直後のデータをアドレスとするデータを扱うアドレシングモードです。

```
MOV     AX, 1234H
MOV     AX, [1234H]
```

この2つの違いをよく理解してください。多くのアセンブラでは, あるアドレスの内容を

[　　]

で表します。[1234H] は 1234H ではなく,「アドレス 1234H の内容」を表すのです。CP/M-86 の ASM86 では [1234H] の代わりに

.1234H

を使いますから注意してください。

レジスタ間接・アドレシング

```
MOV     AX, [BX]
```

これは AX レジスタに BX レジスタの内容をアドレスとするメモリのデータを転送します。AX レジスタに BX レジスタの内容を転送するのではないことに注意してください。[BX] ですから。

レジスタ間接・アドレシングは指定したレジスタの内容をアドレスとするメモリのデータを扱うアドレシングモードです。次の2つ

```
MOV     AX, BX
MOV     AX, [BX]
```

の違いをよく理解してください。CP/M-86 の ASM86 でも [BX] は [BX] です。決して

.BX

ではありません。

シングルインデックスト・アドレシング

MOV　　AX, 1234H [SI]

これは，AXレジスタに1234HとSIレジスタの内容を加えたアドレスにあるメモリの内容を転送します。シングルインデックスト・アドレシングでは，数値と指定されたレジスタの内容を加えてできるアドレスの内容を指すアドレシングモードです。

これは配列のデータを扱うときに使います。いま，

A(0)＝5，A(1)＝6，A(2)＝7，……

として，配列A(I)のバイトデータがアドレス1234Hから

1234H ：5，6，7，……

と並んでいるとします。すると，配列の3番目を取ってくるには，

MOV　　SI, 3
MOV　　AL, 1234H [SI]

とすれば簡単に行えます。

ダブルインデックスト・アドレシング

MOV　　AX, 1234H [BX+SI]

これは1234HとBXレジスタの内容とSIレジスタの内容を加えたアドレスにあるメモリの内容を，AXレジスタに転送します。ダブルインデックストはシングルインデックストのレジスタが2つになったと考えればよいでしょう。

これは，構造体を扱うときなどに使われます。

スタック・アドレシング

PUSH　AX ――――――――⑤

これはAXレジスタの内容をスタックに積む動作をします。スタックとは前にも述べたように，データを一時記憶しておくエリアで，実際にはスタックセグメントのスタックポインタの指し示すメモリに記憶されます。⑤の動作は，

SUB　　SP, 2
MOV　　SS: [SP], AX ――――⑥

と同じといえます。実際には⑥のような命令はありませんが，このようにスタ

ック・アドレシングとは，扱うデータをスタックで指し示すアドレシングモードだといえます。

アドレシングモードの分け方にはいろいろあって，各アドレシングモード名もかなりあります。レジスタやイミディエイト，スタックなどのアドレシングモードは覚えるまでもないでしょう。ほかのモードは，

$$\underbrace{1234H}\left[\underbrace{\begin{Bmatrix}BX\\ \text{or}\\ BP\end{Bmatrix}}+\underbrace{\begin{Bmatrix}SI\\ \text{or}\\ DI\end{Bmatrix}}\right]\text{———⑦}$$

とまとめることができます。これはダブルインデックストの形ですが，ほかのアドレシングモードでは，『〰』部が一部省略されていると考えるわけです。

アドレシングモードの名称を覚えるより，

$$1234H\left[\begin{Bmatrix}BX\\ \text{or}\\ BP\end{Bmatrix}+\begin{Bmatrix}SI\\ \text{or}\\ DI\end{Bmatrix}\right]$$

で覚えるほうがよいでしょう。

アドレシングモードでもう1つ重要なことは，セグメントの問題です。前に，セグメント・プレフィックス（CS:，DS:，ES:，SS:）を省略した場合，たいていデータセグメントが選ばれると述べましたが，BPレジスタ間接のときはスタックセグメントが選ばれます。

```
MOV     AX, [BP]
```

や

```
MOV     AX, [BP+SI]
MOV     AX, 1234H [BP+SI]
```

などは，すべてスタックセグメント内のメモリが対象となります。BPレジスタを間接アドレシングで使うときは，常に注意が必要です。

オペコード表の読み方 10

では，いよいよオペコード表の読み方を説明しましょう。アセンブリ言語のプログラムは，

MOV	AX，1234H
オペコード	オペランド

というように，**オペコード**と**オペランド**に分けられます（オペランドはない場合もある。NOP，HLT など）。

オペコード表には，8086のオペコード，どんなオペランドが使えるか，機械語，必要なバイト数，実行するのにかかる時間，動作，変化するフラグなど，必要なことはすべて記載されています。

それでは，

MOV　　AL，1234H

に関して知りうる限りの情報をオペコード表より読んでみましょう。このオペコード表は ABC 順にソートされています。まず，オペコードの欄を見て MOV を探します。次にオペランドを見ます。いまオペランドは，

AX，1234H

ですから，レジスタとワード長のイミディエイトデータです。

したがって，

reg，imm

のところを見ます。すると，オペレーションコードは，

7 6 5 4 3 2 1 0
1 0 1 1 W reg

となっています。Wはワードのとき 1，バイトのとき 0 となりますが，いまはワード長ですから，Wは 1 です。次に reg は AX レジスタを使っていますから 000となります。したがって，オペレーションコードは，

10111000＝B8H

となります。あとイミディエイトデータ1234Hがありますから，例によって前後を入れ替えて，

34 , 12

つまり,

MOV　　AX, 1234H

は

B8 , 34, 12

となるのです。このように，アセンブリ言語を機械語にオペコード表を見ながら変換できます。ただ，いまやったように時間もかかるし，間違える可能性も大きいので，アセンブラに変換させるのが普通です。バイト数は2-3とありますが，バイトデータのときは2，ワードデータのときは3になるという意味です。いまのは,

B8 , 34, 12

と確かに3バイトとなっています。

次のクロック数は実行に要する時間といえます。4クロックとなっていますから，PC-9801のクロック周波数を4.91MHzとすると,

$4/(4.91\times10^6)\fallingdotseq 815$(n秒)

の時間がかかることになります。ただしこれは理想値で，DRAMのリフレッシュの時間は含まれません。しかも，メモリが十分高速でウエイトがかけられていない，そして偶数アドレスから開始されている，8086のキュー[注13]がいっぱいである，などの条件の下での値です。実際にはこれより遅くなります。

オペレーションは,

(reg)←(data)

となっています。これは，データをレジスタに転送することを表します。

フラグ[注14]は，何も記されていないので,

MOV　　AX, 1234H

はフラグに何の影響も与えないことが分かります。

このように，オペコード表からはかなりの情報が得られます。普通，アセンブリ言語でプログラムする場合,

オペレーションコード

バイト数

は，まったく見る必要がありません。これらは，アセンブラが勝手に作ってくれます。

いちばんよく見るのは，

オペランド

フラグ

です。この命令ではどんなオペランドが使えるのか，またフラグは何が影響を受けるかはとても重要なことです。前にも述べた

MOV　　DS，0A800H

が許されるかどうかは，オペコードのMOVの項で，オペランドとして

sreg，imm

が許されるかどうかで分かります（sreg は segment reg の意)。確かにこのオペランドはありません。したがって，

MOV　　DS，0A800H

は許されません。また

INC　　AX

でキャリーフラグが変わるかどうかは，オペコードのINCの項のフラグ欄を見れば分かります。フラグのCの欄はブランクになっていますから，Cフラグはまったく影響を受けません。

オペランドとフラグ以外ではクロック数を見ることもあります。アセンブリ言語でも，速度が問題になるときにクロック数を考えます。たとえば，

ADD　　AX，1 ―――――⑧

INC　　AX ―――――⑨

の2つはともに，AXレジスタに1を加える命令です。⑧のクロック数は，オペコードADDのオペランド

reg，imm

のクロック数を見れば分かります。4クロックですね。⑨は同じように，オペコードINCのオペランド

reg16

のクロック数より2クロックです。つまり，

ADD　　AX，1

とするよりは

INC　　AX

のほうが速いことが分かります。

このような細かいところまで意識する必要はあまりありませんが，非常に頻繁に通る部分ではけっこう実行時間に差が出てきます。

オペレーションコードは先ほども述べたように，見ることはまずありません。オペコード表なのにオペレーションコードを参照することがないのは妙な気もしますが。

8ビットCPUで，しかも数百バイト程度のプログラムしか書かなかったころならともかく，アドレシングモードもオペコードもかなり増加した16ビットCPUで，数十Kバイトのプログラムもめずらしくなくなった最近では，オペレーションコード表を見ていちいち機械語に翻訳することはまずありません。オペレーションコードが必要なのは，

アセンブラやコンパイラを作る人

アセンブラを持っていない人

に限られるでしょう。

アセンブリ言語でプログラミングしたい方には，アセンブラは不可欠でしょう。

オペレーションコード表の説明はこのくらいにしておきます。どうしても手でアセンブルしたい方は，識別子の表とオペコード表をにらめっこしてアセンブルしてください。

アセンブリ言語の例

解説ばかりではおもしろくないので，また簡単なプログラム例を載せておきます。

「1＋2＋…＋100を計算し，結果を AX レジスタに入れよ」

BASIC では次のようになるでしょう。

```
100  SUM=0
110  C=0
120  C=C+1
130  SUM=SUM+C
140  IF C<>100 GOTO 120
150  END
```

これをそのままプログラミングすると，次のようになります。SUM を AX レジスタ，C を CX レジスタでプログラムした例です。

```
              MOV     AX, 0
              MOV     CX, 0
MAIN_LOOP:
              ADD     CX, 1
              ADD     AX, CX
              CMP     CX, 100
              JNE     MAIN_LOOP
              END
```

解説していきましょう。

```
MOV     AX, 0
MOV     CX, 0
```

で，AX，CX 各レジスタを 0 にします。次の

```
MAIN_LOOP:
```

というのは初めて出てきましたが，これは**ラベル**と呼ばれます。BASIC では，120～140行が繰り返しで，140行から120行に飛んでいます。BASIC の場合，飛び先を行番号で示します。アセンブリ言語でも GOTO 文に相当する**ジャン**

プ命令がありますが，飛び先を指定しようにも，行番号がありません。そこで行番号の代わりに，ラベルを使います。PC-9801 の BASIC ではラベルも使えるので，分かりやすいと思います。ラベルは，アルファベットと@，_が使えるのが普通です。ラベルは

MAIN_LOOP:

のように，ラベル名の最後に：（コロン）をつけて表します。モニタのAコマンドでは，ラベルが使えませんので注意してください。

次に

ADD　　CX，1

で，CX レジスタに 1 を加えます。これは，

INC　　CX

と書いてもよいでしょう。そして，

ADD　　AX，CX

で，AX レジスタに CX レジスタを加えます。つまり，BASIC なら

AX＝AX＋CX

です。

CMP　　CX，100

は CMP 命令で，比較を行います。上の例では，

CX－100

を考えます。この演算の結果，フラグがセット，リセットされるのです。

CMP　　AX，100

ではフラグが変化するだけで AX レジスタはまったく変化しません。

JNE　　MAIN_LOOP

の JNE は条件ジャンプ命令で，フラグの状態により，ジャンプしたりジャンプしなかったりします。JNE は Jump if Not Equal の意味です。つまり

CMP　　AX，100
JNE　　MAIN_LOOP

では，AX が100でなければ MAIN_LOOP へ飛ぶ，という意味になります。このように，CMP と条件ジャンプはたいてい対にして使います。

これをアセンブルすると，

B8，00，00，B9，00，00，83，C1，

01, 01, C8, 83, F9, 64, 75, F6

となります。次にアセンブルの方法を説明しましょう。

N88-DISK BASIC モニタ

MON ⏎

C1800 ⏎

A0 ⏎

として，リスト1-6の右側を打ち込んでください。

リスト1-6

```
0000 B80000            MOV     AX,0000
0003 B90000            MOV     CX,0000
0006 83C101            ADD     CX,0001
0009 01C8              ADD     AX,CX
000B 83F964            CMP     CX,0064
000E 75F6              JNE     0006
```

CP/M-86

ED やほかのエディタでリスト1-7のソースファイルを作ります。1～3行の；以下はコメント，4行の

CSEG

は

「以下がコードセグメントに属す」

ことを宣言しています。

リスト1-7

```
;
;  1+2+3+..+100 ( CP/M-86 )
;
        CSEG

        MOV     AX,0
        MOV     CX,0
MAIN_LOOP:
        ADD     CX,1
        ADD     AX,CX
        CMP     CX,100
        JNE     MAIN_LOOP

        END
```

このソースのファイル名をたとえば,

OHPC.A86

(エクステンションはA86)

とすると,

ASM86 OHPC $HZ SZ ⏎

でアセンブルされ，リスト1-8の

OHPC.LST

ができます。

リスト1-8

```
CP/M ASM86 1.1  SOURCE: LIST2.A86
                                     PAGE   1


                              ;
                              ;  1+2+3+..+100 ( CP/M-86 )
                              ;
                                      CSEG

 0000 B80000                          MOV      AX,0
 0003 B90000                          MOV      CX,0
                              MAIN_LOOP:
 0006 83C101                          ADD      CX,1
 0009 03C1                            ADD      AX,CX
 000B 83F964                          CMP      CX,100
 000E 75F6            0006            JNE      MAIN_LOOP

                                      END


END OF ASSEMBLY.  NUMBER OF ERRORS:   0.  USE FACTOR:  0%
```

MS-DOS

まず，リスト1-9のファイルをEDLINなどで作ります。このファイル名を

OHPC.ASM

(エクステンションはASM)

としておきます。

リスト1-9

```
;
;  1+2+3+..+100 ( MS-DOS )
;
```

```
EXAMPLE SEGMENT BYTE              EXAMPLEというセグメント
        ASSUME  CS:EXAMPLE        EXAMPLEはコードセグメントであることを指定

        MOV     AX,0
        MOV     CX,0
MAIN_LOOP:
        ADD     CX,1
        ADD     AX,CX
        CMP     CX,100
        JNE     MAIN_LOOP

EXAMPLE ENDS                      EXAMPLEセグメントの終わり
        END
```

アセンブルは

MASM OHPC, NULL, OHPC, NULL

で行い，リスト1-10のファイル

OHPC.LST

ができ上がります。

リスト1-10

```
The Microsoft MACRO Assembler              11-08-85      PAGE     1-1


                                ;
                                ;  1+2+3+..+100 ( MS-DOS )
                                ;
0000                            EXAMPLE SEGMENT BYTE
                                        ASSUME  CS:EXAMPLE

0000  B8 0000                           MOV     AX,0
0003  B9 0000                           MOV     CX,0
0006                            MAIN_LOOP:
0006  83 C1 01                          ADD     CX,1
0009  03 C1                             ADD     AX,CX
000B  83 F9 64                          CMP     CX,100
000E  75 F6                             JNE     MAIN_LOOP

0010                            EXAMPLE ENDS
                                        END

 The Microsoft MACRO Assembler              11-08-85      PAGE     Symbols
                                 -1


Segments and groups:

                N a m e                 Size    align   combine class

EXAMPLE. . . . . . . . . . . . .        0010    BYTE    NONE

Symbols:
```

```
                 N a m e                      Type    Value   Attr

MAIN_LOOP. . . . . . . . . . . . .            L NEAR  0006    EXAMPLE

Warning Severe
Errors  Errors
0       0
```

これを実行するには，モニタモードで行います。

MON ⏎

C1800 ⏎

のあと

S0 ⏎

として打ち込んでください。

プログラムは0H～0FH番地ですから，

G0，10 ⏎

で0番地より実行し，10H番地で停止させてください。

何をやったかというと，

$1+2+\cdots+100$

の結果をAXレジスタに求めたのです。レジスタの内容はXコマンドで見ることができます。

XAX ⏎

とすると

AX　13BA

と出るでしょう。

$$13BAH=1\times16^3+3\times16^2+11\times16+10$$
$$=5050$$

ですから，確かに1～100を足した結果がAXレジスタに入っています。

アセンブリ言語を理解するには，自分で短いプログラムを組んで，何度も暴走させるのが近道だと思います。最初は長いプログラムを作ろうとせず，簡単で短いプログラムを作ることを勧めます。長いプログラムでは思ったように動作しない場合，どこに間違いがあるか発見するのが大変です。機械語はフラグ1つ違っても正常に動作しませんから。われわれも一度に大きなプログラムを作るわけではありません。できるだけ小さなモジュールに分けて，各モジュー

ルごとにデバッグするのが普通です。アセンブリ言語を始めたばかりなのですから，動かない大きなプログラムを1つ作るよりも，小さなプログラムを数多く作るほうがよいと思います。

第2章
8086の命令

これからは8086の各命令を解説していきます。あとから引けるように，アルファベット順になっています。

コマンドメニュー1（AAA～CMC）

AAA（ASCII Adjust for Addition）

アンパック BCD（ASCII）の加算補正

コード：00110111＝37H

フラグ：変化 AF, CF

クロック数：4

AAA は AL レジスタに入っている2つのアンパック形式の数値を加算した結果を AX レジスタにアンパック形式の BCD に直して格納します。

BCD とは，Binary Coded Decimal の略で，2進化10進数と呼ばれています。4ビットで表される数値は0～15ですが，BCD ではこのうちの0～9を使って10進数を表す方法です。8ビットのレジスタは普通の2進数なら，

0～255

の数値で表しますが，BCD では

00～99

しか表せません。このように，8ビットのレジスタに2桁の BCD を入れたものをパック形式 BCD と呼びます。これに対して，8ビットのレジスタに1桁の BCD を入れたものを，**アンパック形式BCD** と呼びます。AAA ではこのアンパック形式 BCD を扱います。

いま，

19＋3

を考えてみましょう。19はアンパック形式 BCD で 0109H，3はアンパック形式 BCD で03Hです。AX レジスタに0109H，BL レジスタに03Hを入れる場合，次のようになります。

```
MOV   AX, 0109H
MOV   BL, 03H
```

ADD　AL, BL

AAA

3行目の操作を行った時点で，AXレジスタには010CHが格納されているはずです。ところが，010CHはアンパックのBCDではありません。

ここで，

AAA

を行えば，AXレジスタの値はめでたく0202Hとなります（0202HはアンパックBCDで22の意)。

つまり，バイナリのデータもアンパックBCDのデータも，ともに同じADDを使って加算できますが，アンパックBCDの場合は，その直後に（厳密にはフラグが変化する前に）AAAを行わなければならないのです。

8086はこのようなBCD演算用の命令を数多くもっています。BCD演算は商業用プログラムで必要となります。普通のバイナリで行われますが，10進数でバイナリに変換するときにまるで誤差が出てしまいます。科学技術計算ではあまり問題になりませんが，商業，特に金銭にかかわるプログラムでは1円の誤差も許されません。こういうときにBCD演算を使います（10進数からBCDへの変換時には誤差はまったく出ません)。

もっとも，アセンブラで商業用のプログラムを作ることはないと思いますので，8086のBCD演算命令を使ってプログラムする人は，コンパイラを作る人などに限られると思います。

AAD（ASCII　Adjust for Division）

アンパックBCD（ASCII）の除算補正

コード：11010101　00001010＝D50AH

フラグ：変化 PF, SF, ZF

クロック数：60

AADはアンパック形式BCDどうしの除算のための補正を行います。AADで注意することは〝演算の前″に行わなければならないことです。また，被除数はAXレジスタに入れます。

12÷3

を考えてみましょう。12はアンパック形式BCDで0102H, 3は03Hですから,

```
MOV   AX, 0102H
MOV   BL, 3
AAD
DIV   BL
```

3行目のAADによってAXレジスタの値は000CHとなります。これは,アンパック形式BCDを下のようにバイナリに直したものです。

アンパックBCD　10進　16進
　0102H　→　12　→　0CH

AADはアンパック形式BCDをバイナリに変換するものです。いい換えれば,

AL=AH ＊10+AL
AH=0

という作業をします。

4行目のDIVは除算を行う命令です。くわしくはDIVのところで説明しますが,

```
DIV   BL
```

は，AXレジスタの値をBLレジスタの値で符号なし2進数として除算します。これにより，ALレジスタに09Hが入っているはずです。

AAM (ASCII Adjust for Multiply)

アンパック BCD (ASCII) の乗算補正

コード：11010100　00001010=D40AH

フラグ：変化 PF, SF, ZF

クロック数：83

AAMはAL内のデータをアンパック形式BCDどうしの乗算結果とみなし，アンパック形式BCDに変換します。

AAMで注意することは乗算の後でAAMを実行する点です。

4 × 5

をアンパックBCDで演算することを考えます。

4 →04H, 5→05H

ですから，

```
MOV   AL, 04H
MOV   BL, 05H
MUL   BL
AAM
```

となります。MUL は乗算命令です。くわしくは MUL の項で説明しますが，3行目では，AL レジスタの内容をバイナリデータとみなして乗算し，AX レジスタに格納します。この時点で AX レジスタには

04×06＝18H

が入っています。ここで AAM 命令を実行すれば，

AX＝0204H

となります。つまり，こうです。

16進		アンパック BCD		10進
18H	→	0204H	→	24

AAM は，バイナリデータをアンパック BCD に変換しているといえます。

AAS (ASCII Adjust for Subtraction)

アンパック BCD(ASCII) の減算補正

コード：00111111＝3FH

フラグ：変化 AF，CF

クロック数：4

AAS は AL レジスタのデータをアンパック形式 BCD どうしの減算の結果とみなし，アンパック BCD に変換します。AAS で注意することは，減算の直後に行わなければならないことです。

12－3

をアンパック BCD で計算することを考えます。

10進		アンパック BCD	10進		アンパック BCD
12	→	0102H	3	→	03H

ですから，

```
MOV   AX, 0102H
```

MOV　BL, 03H

SUB　AL, BL

AAS

となります。3行目の操作で，ALとBLレジスタの内容をバイナリデータとみなして減算します。それをAASで補正すると

AX＝0009H

となります。

AAA, AAD, AAM, AASなどのBCD演算用の命令は特殊なことをする人以外は使いませんので，「そんな命令もあるんだ」といった程度でよいと思います。BCD演算を行いたければ，COBOLや商業用BASICなどのコンパイラを通して行えばよいことで，アセンブラで組む必要はないでしょう。

ADC (Add with Carry)

キャリーを含む加算

フラグ：変化 AF, CF, OF, PF, SF, ZF

ADCは加算するときにキャリーも同時に加算します。キャリーフラグは桁上がりが生じたときに立ちます。たとえば

MOV　AX, 0FFFFH

ADD　AX, 2

とすると，下のようになります。

```
      0FFFFH
  +)      2H
  ----------
      10001H
```

そして，

AX＝0001H, キャリーフラグ＝1

となります。

ADC　AX, 3

などは，キャリーが立っていなければ，

AX＝AX＋3

ですが，キャリーが立っていると

AX＝AX＋3＋1

と1が余分に加算されます。一見，妙な機能に思えますが，これは次のような場合，特に便利です。

6789ABCDH＋11111111H

を計算するような場合です。8086のレジスタは16ビット長です。そのため，上の計算を

MOV　AX，6789ABCDH

ADD　AX，11111111H

とすることはできません。6789ABCDHを上位，下位に分けて

MOV　AX，6789H

MOV　BX，ABCDH

として，AXとBXレジスタのペアで表します。あとは筆算と同じように

ADD　BX，1111H

ADD　AX，1111H

とすればよいのです。1行目実行の時点で，桁上がりがあるかもしれませんし，ないかもしれません。もし8086にADC命令がなければ

桁上がりがない→ AX＝AX＋1111H

桁上がりがある→ AX＝AX＋1111H＋1

とプログラムを分けなければなりません。

ところが，ADCがあれば一度に行えます。

このようにADCを使えば多倍長(1000桁でも10000桁でもメモリの許す限り）の加算を行うことができます。

ADD（Addition)

加算

フラグ：変化 AF，CF，OF，PF，SF，ZF

ADDは加算を行います。もうすでにおなじみの命令の1つでしょう。

8086で非常に助かるのは，多くの演算がメモリに対して直接行える点です。データセグメントの1234H番地のバイトデータに3を加えるには，

ADD　BYTE　[1234H]，3

とすることができます。8ビットCPUをやってきて，8086は初めてという人は，

```
MOV    AL, [1234H]
ADD    AL, 3
MOV    [1234H], AL
```

などとやっていますが，これは1つの命令で行えます。

AND（AND：logical conjunction）
論理積
フラグ：変化 CF, OF, PF, SF, ZF

ANDは論理積をとります。BASICのANDと同様です。特に説明することもないでしょう。

CALL（Call a procedure）
サブルーチンコール
フラグ：変化なし

CALL命令はBASICのGOSUBに相当します。アセンブリ言語でもサブルーチンが使えます。AXレジスタを0にするサブルーチンを

CLEAR__AX

としましょう。

CLEAR__AX

は次のようになります。

```
CLEAR__AX:MOV   AX, 0
          RET
```

1行目の

CLEAR__AX:

はラベルです。N_{88}-BASICでは

*CLEAR.AX

とでも書くところでしょう。最後のRETは，サブルーチンから返る命令で，

BASIC の RETURN に相当します。このサブルーチンを呼ぶには，プログラムの中で

　CALL　CLEAR_AX

と記述します。

CALL には

　セグメント内コール

　セグメント外コール

の 2 つがあります。通常はセグメント内コールを使います。セグメント内コールでは同一コードセグメント内にあるサブルーチンに飛ぶのに使います。64K 以上離れた別のセグメントをコールするには，セグメント外コールを使います。モニタの A，L コマンドでは，セグメント外コールは使えません。CP/M-86 の ASM86 ではセグメント内を CALL，セグメント外を CALLF（Call Far）と書いて区別します。

セグメント内とセグメント外の違いは，スタックにもどりオフセットだけを積むか，もどりセグメントも積むかの違いです。

RET も CALL に対応して，セグメント内，セグメント外の区別があります。モニタではセグメント外リターンはありません。

CP/M-86 の ASM86 では，セグメント内が RET，セグメント外が RETF（―Far），MS-DOS の MASM では，セグメント内は RET または RET　NEAR，セグメント外は RET　FAR と書くことになっています。

当面はセグメント内だけでしょうから，コールは CALL，リターンは RET だけを覚えておけばよいでしょう。

モニタの A コマンドではラベルが使えませんので，

　CALL　1234

というふうに直接，サブルーチンのアドレスを書いて使います。

また，CALL には間接コールというものがあります。たとえば，1234H のサブルーチンを呼ぶときに

　CALL　1234H

とするのではなく

　MOV　AX，1234H

　CALL　AX

というふうに，レジスタの内容を飛び先とする方法です。

レジスタの代わりにメモリ間接で飛ぶこともできます。メモリ間接の場合，セグメント外コールも行えます。コールするときどんなオペランドが可能かは，オペレーションコード表の，オペランド項を見ればすぐ分かります。

たとえば

CALL　WORD PTR [BX] [SI]

なども許されることが分かるでしょう。これは，BX レジスタの内容と SI レジスタの内容を加えたアドレスのメモリ内容が指すところにコールします。

CBW (Convert Byte to Word)

バイトからワードへの変換

コード：10011000＝98H

フラグ：変化なし

クロック数：2

CBW は AL レジスタのバイトデータを AX レジスタのワードに変換します。

これは，AL レジスタが負ならば

AH＝FFH

正ならば

AH＝00H

と動行します。

CBW は除算の前に行うことがよくあります。たとえば，

15÷3

を行うには，

```
MOV   AL, 15
CBW
MOV   BL, 3
DIV   BL
```

とします。4 行目では，AX レジスタを BL レジスタで割りますから，割られる数が AL レジスタに入っている場合，CBW を使って AL レジスタのデー

タをAXレジスタのワードデータに変換しなくてはなりません。

これはよく間違えることなので注意が必要です。6809には SEX(Sign EXtend) 命令という気のきいた命令がありましたが，CBWはSEX命令に相当します。

CLC (Clear Carry flag)

キャリーのクリア

コード：11111000＝F8H

フラグ：変化 CF

クロック数：2

CLC命令はキャリーフラグを0にします。逆に1にするにはSTC(SetCarry flag) を使います。CLC，STC命令は，サブルーチンの返す値が1か0しかないような場合，キャリーを使って返すときなどによく使います。

CLD (CLear Direction flag)

ディレクションフラグのクリア

コード：11111100＝FCH

フラグ：変化 DF

クロック数：2

CLDはDF（ディレクションフラグ）を0にするときに使います。

DF＝0

のときには，ストリング命令時にオートインクリメントモードとなります。

CLI (CLear Interrupt flag)

割り込みフラグのクリア

コード：11111010＝FAH

フラグ：変化 IF

クロック数：2

CLIはインタラプトを禁止します。したがって，スタックポインタの設定，割り込みベクトルの書き換えなど，インタラプトの起こってほしくないときに使います。インタラプトを禁止する必要がなくなれば，ただちにインタラプト許可(STI)を実行しなければなりません。なお，NMI（ノンマスカブルインタラプト）はCLIでは禁止できません。最初のうちはCLIはまったく必要ないでしょう。

CMC（CoMplement Carry flag)

キャリーの反転

コード：11110101＝F5H

フラグ：変化 CF

クロック数：2

CMCはキャリーフラグを1→0，0→1と反転させます。

簡単なプログラム

命令の説明ばかりではつまらないので，アセンブリ言語による簡単なプログラム例を見てみましょう。画面消去プログラムです。

PC-8801では高速画面消去プログラムが話題になっていましたが，それをPC-9801で書いてみましょう。

CP/M-86

エディタでリスト2-1を打ち込み，

OHPC．A86

というファイル名にします。

リスト2-1

```
;
; CLEAR GRAPHIC V-RAM ( CP/M-86 )
;
        CSEG
        ORG     000H
```

```
CLS:
        MOV     AX,0A800H
        CALL    CLEAR_ONE_PLANE
        MOV     AX,0B000H
        CALL    CLEAR_ONE_PLANE
        MOV     AX,0B800H
        CALL    CLEAR_ONE_PLANE
        IRET
CLEAR_ONE_PLANE:
;
; SUBROUTINE CLEAR_ONE_PLANE
; ARGUMENT AX -- SEGMENT
;
        CLD
        MOV     ES,AX
        MOV     CX,4000H
        MOV     DI,0
        MOV     AX,0
        REP     STOSW
        RET

        END
```

ASM86 OHPC $SZ HZ ⏎

でリスト2-2の

OHPC. LST

ができ上がります。

リスト2-2

```
CP/M ASM86 1.1  SOURCE: LIST1.A86
                                 PAGE    1


                          ;
                          ; CLEAR GRAPHIC V-RAM ( CP/M-86 )
                          ;
                                  CSEG
                                  ORG     000H
                          CLS:
0000 B800A8                       MOV     AX,0A800H
0003 E80D00        0013           CALL    CLEAR_ONE_PLANE
0006 B800B0                       MOV     AX,0B000H
0009 E80700        0013           CALL    CLEAR_ONE_PLANE
000C B800B8                       MOV     AX,0B800H
000F E80100        0013           CALL    CLEAR_ONE_PLANE
0012 CF                           IRET
                          CLEAR_ONE_PLANE:
                          ;
                          ; SUBROUTINE CLEAR_ONE_PLANE
                          ; ARGUMENT AX -- SEGMENT
                          ;
0013 FC                           CLD
0014 8EC0                         MOV     ES,AX
0016 B90040                       MOV     CX,4000H
0019 BF0000                       MOV     DI,0
001C B80000                       MOV     AX,0
```

```
 001F F3AB                          REP     STOSW
 0021 C3                            RET

                                    END


END OF ASSEMBLY.  NUMBER OF ERRORS:    0.  USE FACTOR:   0%
```

MS-DOS

エディタでリスト2-3を打ち込み,

OHPC. ASM

というファイル名にします。

リスト2-3

```
;
; CLEAR GRAPHIC V-RAM ( MS-DOS )
;
EXAMPLE SEGMENT BYTE
        ASSUME  CS:EXAMPLE
        ORG     0000H
CLS:
        MOV     AX,0A800H
        CALL    CLEAR_ONE_PLANE
        MOV     AX,0B000H
        CALL    CLEAR_ONE_PLANE
        MOV     AX,0B800H
        CALL    CLEAR_ONE_PLANE
        IRET
CLEAR_ONE_PLANE:
;
; SUBROUTINE CLEAR_ONE_PLANE
; ARGUMENT AX -- SEGMENT
;
        CLD
        MOV     ES,AX
        MOV     CX,4000H
        MOV     DI,0
        MOV     AX,0
        REP     STOSW
        RET

EXAMPLE ENDS
        END
```

MASM OHPC. NUL, OHPC, NUL ⏎

でリスト2-4の

OHPC. LST

を得ます。

リスト2-4

```
The Microsoft MACRO Assembler            11-08-85      PAGE    1-1


                                   ;
                                   ; CLEAR GRAPHIC V-RAM ( MS-DOS )
                                   ;
0000                               EXAMPLE SEGMENT BYTE
                                           ASSUME  CS:EXAMPLE
0000                                       ORG     0000H
0000                               CLS:
0000  B8 A800                              MOV     AX,0A800H
0003  E8 0013 R                            CALL    CLEAR_ONE_PLANE
0006  B8 B000                              MOV     AX,0B000H
0009  E8 0013 R                            CALL    CLEAR_ONE_PLANE
000C  B8 B800                              MOV     AX,0B800H
000F  E8 0013 R                            CALL    CLEAR_ONE_PLANE
0012  CF                                   IRET
0013                               CLEAR_ONE_PLANE:
                                   ;
                                   ; SUBROUTINE CLEAR_ONE_PLANE
                                   ; ARGUMENT AX -- SEGMENT
                                   ;
0013  FC                                   CLD
0014  8E C0                                MOV     ES,AX
0016  B9 4000                              MOV     CX,4000H
0019  BF 0000                              MOV     DI,0
001C  B8 0000                              MOV     AX,0
001F  F3/ AB                               REP     STOSW
0021  C3                                   RET

0022                               EXAMPLE ENDS
                                           END
 The Microsoft MACRO Assembler            11-08-85      PAGE    Symbols
                                    -1


Segments and groups:

                N a m e                    Size    align   combine class

EXAMPLE. . . . . . . . . . . . .           0022    BYTE    NONE

Symbols:

                N a m e                    Type    Value   Attr

CLEAR_ONE_PLANE. . . . . . . . .           L NEAR  0013    EXAMPLE
CLS. . . . . . . . . . . . . . .           L NEAR  0000    EXAMPLE

Warning Severe
Errors  Errors
0       0
```

モニタ

BASIC レベルから

MON ⏎

]C1C00 ⏎

]A0 ⏎

のあと，リスト2-5の右部分を打ち込み，

CTLR-B

で BASIC に抜けます。そして，

DEF SEG=&H1C00 ⏎

A=0：CALL A ⏎

で実行できます。

リスト2-5

```
0000 B800A8         MOV    AX,A800
0003 E80D00         CALL   0013
0006 B800B0         MOV    AX,B000
0009 E80700         CALL   0013
000C B800B8         MOV    AX,B800
000F E80100         CALL   0013
0012 CF             IRET
0013 FC             CLD
0014 8EC0           MOV    ES,AX
0016 B90040         MOV    CX,4000
0019 BF0000         MOV    DI,0000
001C B80000         MOV    AX,0000
001F F3             REP
0020 AB             STOSW
0021 C3             RET
```

AAA 命令から CMC 命令までを解説しました。各命令がどのようなときに使われるかを中心に説明していきますので，AAA はどんな命令だっけ？と迷ったときなどに見返せば，すぐに分かることと思います。また，プログラミングするときにその命令で注意すべき点もできるだけ説明しているつもりです。説明に出てきた命令を使って，一日も早くプログラミングを始めることを望みます。

言語は使ってみて初めて理解できるものですから，何度も暴走させて身につけてください。

コマンドメニュー 2 (CMP～IMUL)

CMP (CoMPare)

比較

コード：オペコード表参照

フラグ：AF, CF, OF, PF, SF, ZF,

クロック数：オペコード表参照

CMP は，デスティネーションからソース[注15]を減算します。それでは，SUB と変わらないことになりますが，CMP 命令では変化するのはフラグだけです。だから

```
MOV   AX, 1000
CMP   AX, 999
```

および

```
MOV   AX, 1000
SUB   AX, 999
```

の 2 つでは，フラグはまったく同じ変化をしますが，CMP のほうでは AX は 1000のままです。SUB のほうはもちろん

AX＝1

となっています。フラグの変化以外，何の影響もない命令ではあまり意味がないような気がしますが，この命令は非常に大切です。CMP は条件分岐をするときに使われるのです。条件分岐は BASIC では，

IF～THEN

でおなじみだと思います。CMP がいかに重要かは，BASIC で IF～THEN 文のないプログラムはほとんどないことを考えれば分かります。CMP 命令は単独では使われず，たいてい

```
CMP   AX, 99
JE    LABEL 1
```

などと条件分岐命令 (JNE, JNO, JNP, JNS, JO, JP, JS など) とともに使いま

す。CMP 命令によりフラグが変化し，その変化に従って分岐するかどうかが決まります。どのフラグがどのようなときに立つかということは条件分岐命令と組にして使う限り，意識する必要はまったくありません。

AX レジスタが 0 ならば LABEL 1 に飛び，0 以外ならば LABEL 2 を実行するプログラムは，

```
CMP  AX, 0 ―――――――――①
JE   LABEL 1
JMP  LABEL 2
```

となります。これはフラグを含めて考えると次のようになります。

①のところで，AX レジスタと 0 が比較され，その結果

AX=0 → ゼロフラグ=1

AX≠0 → ゼロフラグ=0

となります。ほかのフラグも

AX−0

の結果に従って変化しますが，ここでは無視します。

次の JE (Jump if Equal) 命令は，直前の比較の結果がイコールであればジャンプする命令ですが，実際の動作としては，

「ゼロフラグが 1 ならばジャンプする」

という動作をします。そのため，

JE LABEL 1

は，

ゼロフラグ=1 → LABEL 1 へ

ゼロフラグ=0 → 次の命令へ

となります。ところが，ゼロフラグは，

AX=0→1,

AX≠0→0

となっていますから，結局

AX=0 → LABEL 1へ

AX=1 → LABEL 2へ

という動作を行うわけです。条件関係には=，≠，>，≧，<，≦がありますが，これに従って条件分岐命令は図2-1のように使い分けなければなりません。

図2-1

	符合なし	符合つき
=	JE JZ	JE JZ
≠	JNE JNZ	JNE JNZ
<	JB JNAE	JL JNAGE
≧	JNB JAE	JNL JGE
>	JA JNBE	JG JNLE
≦	JNA JBE	JNG JLE

符号つき，符号なしで命令が違っていることに注意してください。図2-1があればプログラミング上は問題ないと思います。たとえば，

「符号なしで AX≧BX ならば LABEL 1 へジャンプする」

プログラムは，図2-1によれば符号なしの≧が JAE (Jump if Above or Equal) ですから，

```
CMP   AX, BX
JAE   LABEL 1
```

となるわけです。

CMP 命令で比較できるものは，オペレーションコード表を見れば分かりますが，

レジスタ，レジスタ
メモリ，レジスタ
レジスタ，メモリ
メモリ，イミディエイトデータ
レジスタ，イミディエイトデータ
アキュムレータ，イミディエイトデータ

の6つです。どの組み合わせができるかというのは，最終的にはオペレーションコード表を見るしかないのですが，8086で一般的にいえることは，メモリとメモリを同時にソースとデスティネーションに選べない[注16]ということでしょう。それ以外はたいていできると覚えておけば，まず間違いありません。

CMPS (CoMPare String)

メモリとメモリを比較

コード：{10100110＝A6H(バイト)
　　　　{10100111＝A7H(ワード)

フラグ：AF, CF, OF, PF, SF, ZF

クロック数：22

8086にはストリング命令というものがあって8086の大きな特徴の1つとなっています。ストリング命令というのは，メモリ内のデータの移動，代入，比較を効率よく行うための命令群で，Z80でいえば，

LDIR, CPIR

を強力にしたような命令です。8086のストリング命令には，

MOVS　メモリ，メモリ間の移動

LODS　AL/AX レジスタにメモリのデータをロード

STOS　メモリに AL/AX レジスタの内容をロード

SCAS　メモリと AL/AX レジスタの比較

CMPS　メモリとメモリの比較

の5つのストリング・プリミティブ命令があります。これらは単独で用いられることもありますが，たいていリピートプレフィックス（REP, REPE, REPNE, REPNZ, REPZ）を組み合わせて使います。

話をCMPSにもどしましょう。CMPSはメモリとメモリの比較を行います。ソースとデスティネーションは

DS: [SI], ES: [DI]

の2つです。バイトの値の比較ならばCMPSB，ワードならばCMPSWを使います。比較のあとのSI, DIレジスタは自動的に増加あるいは減少します。増加するか減少するかはDF（ディレクションフラグ）によります。

DF＝0　→　増加

DF＝1　→　減少

となります。増減の値はCMPSBならば1，CMPSWならば2です。つまり1回の比較が終わると次のデータの位置をSI, DIレジスタが指してくれている

わけです。次のような具体的な例で考えてみましょう。

「DSセグメントの1000H番地からの256バイトとESセグメントの2000H番地からの256バイトとが一致していれば，LABEL 1に，そうでなければLABEL 2に飛ぶプログラムを書け」

もしCMPS命令がなければ，

```
        MOV   SI, 1000H
        MOV   DI, 2000H
        MOV   CX, 256
    L1:
        MOV   AL, [SI]
        CMP   AL, ES: [DI]
        JE    L2
        JMP   LABEL 2
    L2:
        INC   SI
        INC   DI
        LOOP  L1        注17
        JMP   LABEL 1
```

となりますが，CMPSBを知っていれば，

```
        CLD
                  (オートインクリメントモード)
        MOV   SI, 1000H
        MOV   DI, 2000H
        MOV   CX, 256
    L1:
        CMPSB
        JE    L2
        JMP   LABEL 2      一致しない
    L2:
        LOOP  L1
        JMP   LABEL 1      一致する
```

と，かなり簡単になります。さらに，REPE命令を知っていれば，

```
CLD
MOV     SI, 1000H
MOV     DI, 2000H
MOV     CX, 256
REPE    CMPSB
JE      LABEL 1
JMP     LABEL 2
```

ときわめて簡単になります。

REPEは，CXをカウントダウンし，CXが0でかつCMPSBと等しい間繰り返せの意味になります。REPプレフィックスは，ストリングプリミティブ命令の前でしか使えないことに注意してください。

このようにストリング命令はREP命令と組み合わせて使うことにより，プログラムを非常にコンパクトにすることができます。

ストリング命令で注意することは，ソースやデスティネーションがメモリのときは

ソース：　DS:［SI］
デスティネーション：　ES:［DI］

となることです。SIのことをSource Index Register，DIのことをDestination Index Registerというのは，これに由来しています。セグメント・オーバーライド・プレフィックスによってソース側のデフォルトセグメントをDSからほかのセグメントに変更することも可能です。また，SI，DIレジスタは自動的に変更されますから，ストリング命令を実行する前に必ずDFフラグをセット/リセットすることが必要です（CLD，STD命令)。

ページのプログラムも，実はこのストリング命令を使っています。分からない部分は，

```
REP   STOSW
```

でしょうが，これはCXレジスタをカウントダウンしてCXが0の間，AXレジスタの内容をES:［DI］に転送することを意味します。もしこれを使わなければ，

```
L1:
```

```
MOV   ES: [DI], AX
INC   DI
INC   DI
SUB   CX, 1
JNE   L1
```

と長くなっています。実行時間もクロック数を計算すれば分かりますが，REP STOSW としたほうが速くなっています。ストリング命令はこのように，プログラムを小さく速くすることができます。

CWD (Convert Word to Doubleword)

AX レジスタの符号を DX レジスタに拡張

コード：10011001＝99H

フラグ：変化なし

クロック数：5

CWD 命令は AX レジスタの符号を DX レジスタに拡張します。つまり，

AX＞8000H → DX＝0

その他 → DX＝FFFFH

となります。CBW 命令の16ビット版といえるでしょう。

DX レジスタと AX レジスタで32ビットデータを表すことはよくあります。

たとえば，12345を325で割るときがそうです。これをプログラミングするとすれば，

```
MOV   AX, 12345
CWD
MOV   BX, 325
IDIV  BX
                    (AX に答が入る)
```

となります。IDIV BX は DX: AX を BX で割りますから，必ず CWD 命令を実行して AX の16ビットデータを DX: AX の32ビットデータに変換しなくてはなりません。IDIV 命令を使うときには CWD が必要かどうかをよく考えなくてはなりません。

DAA (Decimal Adjust for Addition)

加算後のアキュムレータの内容を10進数に変換

コード：00100111＝27H

フラグ：AF，CF，PF，SF，ZF が変化，OF 不定

クロック数：4

DAA はパック形式のBCDどうしの加算結果をパック形式BCDに直します。ADDやADCの直後（厳密にはフラグ，データの変化する前に）行います。

たとえば，

12＋19

を計算するときは，

12はパック形式BCDで　12H

19　　　〃　　　19H

ですから，

MOV　L，12H

ADD　AL,19H

DAA

とすればよいのです。

MOV　AL,12H

ADD　AL,19H

を行った時点で，ALレジタには，2BHが入っています。ここで，

DAA

とすれば，

AL＝2BH　→　31H（パック形式BCD）

となり，

12＋19＝31

が計算されます。BCD演算はコンパイラ作成時などには使われますが，アセンブリ言語プログラムで使うことはあまりありません。こんな命令もあるのだくらいに考えておけばよいでしょう。

DAS (Decimal Adjust for Subtraction)

減算後のアキュムレータの内容を10進数に変換

コード：00101111＝2FH

フラグ：AF, CF, PF, SF, ZF が変化，OF が不定

クロック数：4

DAS は，パック形式どうしの減算の結果をパック形式 BCD に直します。DAA の減算用といえるでしょう。

21－3

をプログラミングしてみると，

21 → 21H（パック形式 BCD）

3 → 03H（パック形式 BCD）

ですから，

```
MOV   AL, 21H
DAS
                  (AL に答が入る)
SUB   AL, 03H
```

となります。

```
MOV   AL, 21H
SUB   AL, 03H
```

で，AL は 1EH となります。ここで，

```
DAS
```

とすれば，

AL＝1EH → 18H（パック形式 BCD）

となって，

21－3＝18

を得ます。

DEC (DECrement destination by one)

デクリメント

コード：オペコード表参照 フラグ：AF, OF, PF, SF, ZF が変化 クロック数：オペコード表参照

DEC 命令はオペランドから 1 を引きます。

```
MOV   X, 1000H
DEC   AX
```

はもちろん，

AX＝0FFFH

となります。オペレーションコード表のオペランドを見れば分かるとおり，

```
DEC   ARRAY [BX] [SI]
```

なども許されます。

```
DEC   AX        (2 クロック)
```

は，

```
SUB   AL, 1  (4 クロック)
```

よりも 2 クロック分速くなりますから，1 を引く場合は，

```
DEC   AX
```

とすべきでしょう。

DEC 命令で注意が必要なのは「キャリーフラグが変化しない」点です。SUB 命令ではキャリーフラグは変化しますから，

```
SUB   AX, 1
JB    LABEL 1
```

と，

```
DEC   AX
JB    LABEL 1
```

とは等価ではありません。

```
SUB   AX, 1
JB    LABEL 1
```

のほうは AX が 0 のときには LABEL 1 に分岐しますが，

```
DEC   AX
JB    LABEL 1────────②
```

は，どうなるかは不定です（②の前にキャリーが立っていたかどうかで決まる)。これはINC命令でも同様です。

DIV（DIVide)

符号なし除算

コード：オペコード表参照

フラグ：AF，CF，OF，PF，SF，ZF不定

クロック数：オペコード表参照

DIV命令は下の形式で符号なしの除算を行います。

商　AL　　余り　AH

商　AX　　余り　DX

割られる数はAH:ALの16ビットデータか，DX: AXの32ビットデータです。

1234÷32

は，

```
MOV   AX, 12345
MOV   DX, 0
MOV   BX, 32
DIV   BX
```

となります。DIVで重要なことは，CBW，CWDの命令のところで解説したように，データの型を合わせることでしょう。

また，もう1つ非常に大切なことは，0による除算です。8086では，0による除算が行われると内部インタラプトの0番がかかります。つまり物理アドレスの0000H，0001Hの内容をオフセット，0002H，0003Hの内容をセグメントとするアドレスに制御が移されます。PC-9801のBASIC使用時は，これはROM内のIRET（インタラプトからのリターン）命令を指しており，MS-DOSでは「0で除算しました」のメッセージが出るようになっています。0で割った際の処理ルーチンをユーザー側で作ることもできますが，初めのうちは0で割らないように注意するほうが賢明でしょう。

またDIV命令は非常に時間のかかる命令です。

```
DIV   BX
```

の処理などは，144〜162クロックもかかります。オペランドがメモリだともっとかかります。

SHRなど，シフト命令で置き換えられるところはシフトですますべきです。

DIV は 8 ビット CPU にはない便利な命令ですが，いまの 3 点

(1)型のチェック

(2) 0 による除算

(3)時間がかかる

をよく頭に入れてプログラミングしてください。

ESC (ESCape)

エスケープ

コード：オペコード表参照

フラグ：変化なし

クロック数：オペコード表参照

ESC 命令は8086の命令ではありますが，メモリオペランドをアクセスしてバスにのせる以外何もしません。ESC 命令はほかのプロセッサ，つまり8087に対して命令を発行する際に使われます。

アセンブリ言語で8087をプログラミングすることはまれで，私は二度しか行ったことはありません。だいたい，フローティングポイントの演算を行うプログラムをアセンブリ言語で作ることはほとんど無意味だと思います。多くのCコンパイラ，Pascal などは8087をサポートしているので，コンパイラを使うことを勧めます。

HLT (HaLT)

ホールト

コード：11110100＝F4H

フラグ：変化せず

クロック数： 2

HLT はプログラムの実行を停止します。一度ホールト状態に入ると，抜け

出すにはインタラプトかリセットを行わなければなりません。HLT命令を使うことはまずないと思います（外部インタラプトをうまく使える人は別ですが……）。

IDIV（Integer DIVision signed） 符号つき除算 コード：オペコード表参照 フラグ：AF, CF, OF, PF, SF, ZF 不定 クロック数：オペコード表参照

IDIV命令は符号つき除算を行います。割られる数はAH: ALの16ビットか，DX: AXの32ビットです。結果はDIVと同様

AL　商　AH　余り

AX　商　DX　余り

となります。IDIVで注意することは，DIVと同様に，

(1)型合わせ

(2)0による除算

(3)長い実行時間

です。型合わせには，IDIVが符号つきですからCBWやCWDが最適です。

DIV，IDIVともにいえることですが，イミディエイトデータで割れないことも落とし穴といえるでしょう。

```
IDIV   1234
```

とは書けない点です。

IMUL（Integer MULtiply） 符号つき乗算 コード：オペコード表参照 フラグ：AF, PF, SF, ZF 不定，CF, OF 変化 クロック数：オペコード表参照

IMUL命令は符号つき乗算を行います。かけられる数はALまたはAXで，

結果は AX または DX: AX となります。

IMUL で注意することは，IDIV と同様，

(1)長い実行時間

(2)イミディエイトデータをかけられない

ことがあげられます。特に時間は，

IMUL　BX

で128〜154クロックもかかりますから，シフト命令で置き換えられるならば，置き換えたほうがよいでしょう。

イミディエイトデータがかけられないのは，

IMUL　1234

とできないという意味です。

また，扱っているデータが符号つきか符号なしかを常に頭に入れておくことは重要で，符号つきなら IMUL，IDIV，符号なしなら MUL，DIV を確実に使うべきです。

プログラミングから実行まで

CMP から IMUL の解説を行いました。次にまた，簡単なプログラム例を載せておきます。

「キャラクタの VRAM に0〜255のデータを書き込むプログラム」

です。今回は，CP/M－86上，MS－DOS 上でも実行できるようにしておきました。せっかく CP/M－86や MS－DOS 上でアセンブルしても，実行は DISK BASIC に移ってからというのでは二度手間といえるでしょう。各 DOS の実行ファイルの作り方，抜け方をこのプログラムで研究してください。

CP/M-86

ED などのエディタでリスト2-6を打ち込み，ファイル名を OHPC.A86 とします。そして，

ASM86　OHPC　$SZ　⏎

でリスト2-7の LST ファイルを得ます。

実行は，

リスト2-6

```
;
; OH! PC DEMO PROGRAM ( CP/M-86 )
;
        CSEG
        ORG     100H
        CLD
        MOV     AX,0A000H
        MOV     ES,AX
        XOR     AL,AL
OHPC1:
        CALL    FILL_C_VRAM
        INC     AL
        CMP     AL,0
        JNE     OHPC1

        XOR     CL,CL )
        XOR     DL,DL } CP/M-86から抜ける
        INT     224   )
FILL_C_VRAM:
;
; ARGUMENT AL --> DATA TO BE STORED
;
        XOR     DI,DI
        MOV     CX,80*25*2
        REP     STOSB
        RET

        END
```

リスト2-7

```
CP/M ASM86 1.1  SOURCE: LIST1.A86
                          PAGE    1


                                ;
                                ; OH! PC DEMO PROGRAM ( CP/M-86 )
                                ;
                                        CSEG
                                        ORG     100H
 0100 FC                                CLD
 0101 B800A0                            MOV     AX,0A000H
 0104 8EC0                              MOV     ES,AX
 0106 32C0                              XOR     AL,AL
                                OHPC1:
 0108 E80C00        0117                CALL    FILL_C_VRAM
 010B FEC0                              INC     AL
 010D 3C00                              CMP     AL,0
 010F 75F7          0108                JNE     OHPC1

 0111 32C9                              XOR     CL,CL
 0113 32D2                              XOR     DL,DL
 0115 CDE0                              INT     224
                                FILL_C_VRAM:
                                ;
                                ; ARGUMENT AL --> DATA TO BE STORED
                                ;
 0117 33FF                              XOR     DI,DI
 0119 B9A00F                            MOV     CX,80*25*2
```

```
011C F3AA                           REP      STOSB
011E C3                             RET

                                    END

END OF ASSEMBLY.  NUMBER OF ERRORS:   0.  USE FACTOR:  0%
```

GENCMD OHPC 8080 ⏎

OHPC ⏎

です。

MS-DOS

EDLINなどのエディタでリスト2-8を打ち込み，ファイル名をOHPC.ASMとします。

そして，

リスト2-8

```
;
; OH! PC DEMO PROGRAM ( MS-DOS )
;
STACK1  SEGMENT STACK
        DW      100 DUP (?)
TOS     LABEL   WORD
STACK1  ENDS

CODE    SEGMENT BYTE
        ASSUME  CS:CODE,SS:STACK1
START:
        CLD
        MOV     AX,0A000H
        MOV     ES,AX
        XOR     AL,AL
OHPC1:
        CALL    FILL_C_VRAM
        INC     AL
        CMP     AL,0
        JNE     OHPC1

        MOV     AH,4CH  ⎫MS-DOSから
        INT     21H     ⎬抜ける。Ver 1.25
                        ⎭ではINT 20H
FILL_C_VRAM:
;
; ARGUMENT AL --> DATA TO BE STORED
;
        XOR     DI,DI
        MOV     CX,80*25*2
        REP     STOSB
        RET

CODE    ENDS
        END     START
```

MASM　OHPC, OHPC, OHPC, NUL　⏎

でリスト2-9のファイルを得ます。

実行は,

リスト2-9

```
 The Microsoft MACRO Assembler                  11-08-85       PAGE    1-1



                                    ;
                                    ; OH! PC DEMO PROGRAM ( MS-DOS )
                                    ;
0000                                STACK1  SEGMENT STACK
0000      64 [                              DW      100 DUP (?)
                 ????
                        ]

00C8                                TOS     LABEL   WORD
00C8                                STACK1  ENDS

0000                                CODE    SEGMENT BYTE
                                            ASSUME  CS:CODE,SS:STACK1
0000                                START:
0000  FC                                    CLD
0001  B8 A000                               MOV     AX,0A000H
0004  8E C0                                 MOV     ES,AX
0006  32 C0                                 XOR     AL,AL
0008                                OHPC1:
0008  E8 0015 R                             CALL    FILL_C_VRAM
000B  FE C0                                 INC     AL
000D  3C 00                                 CMP     AL,0
000F  75 F7                                 JNE     OHPC1

0011  B4 4C                                 MOV     AH,4CH
0013  CD 21                                 INT     21H

0015                                FILL_C_VRAM:
                                    ;
                                    ; ARGUMENT AL --> DATA TO BE STORED
                                    ;
0015  33 FF                                 XOR     DI,DI
0017  B9 0FA0                               MOV     CX,80*25*2
001A  F3/ AA                                REP     STOSB
001C  C3                                    RET

001D                                CODE    ENDS
                                            END     START
 The Microsoft MACRO Assembler                  11-08-85       PAGE    Symbols
                                    -1



Segments and groups:

                N a m e                     Size    align   combine class

CODE . . . . . . . . . . . . . .            001D    BYTE    NONE
STACK1 . . . . . . . . . . . . .            00C8    PARA    STACK
```

```
Symbols:

                Name                        Type    Value   Attr

FILL_C_VRAM. . . . . . . . . . . .          L NEAR  0015    CODE
OHPC1. . . . . . . . . . . . . . .          L NEAR  0008    CODE
START. . . . . . . . . . . . . . .          L NEAR  0000    CODE
TOS. . . . . . . . . . . . . . . .          L WORD  00C8    STACK1

Warning Severe
Errors  Errors
0       0
```

LINK　OHPC; ⏎

OHPC　⏎

です。

DISK BASIC

MON　⏎

C1F 00　⏎

のあと，

A0　⏎

として，リスト2-10を打ち込みます。

リスト2-10

```
0000 FC              CLD
0001 B800A0          MOV     AX,A000
0004 8EC0            MOV     ES,AX
0006 32C0            XOR     AL,AL
0008 E80B00          CALL    0016
000B FEC0            INC     AL
000D 3C00            CMP     AL,00
000F 75F7            JNE     0008
0011 32C9            XOR     CL,CL
0013 32D2            XOR     DL,DL
0015 CF              IRET
0016 31FF            XOR     DI,DI
0018 B9A00F          MOV     CX,0FA0
001B F3              REP
001C AA              STOSB
001D C3              RET
```

CTRL-B

でBASICへ抜け，

DEF SEG=&H1F0: A=0: CALL　A　⏎

で実行します。

ROM BASIC

MON ⏎

C1F 00 ⏎

のあと,

E0 ⏎

としてリスト2-11のダンプを打ち込み,

CTRL-B

でBASICに抜けます。

実行方法はDISK BASICと同じです。

リスト2-11

```
0000 FC B8 00 A0 8E C0 32 C0 E8 0B 00 FE C0 3C 00 75    ク  ■タ2タ♠    タく u
0010 F7 32 C9 32 D2 CF 31 FF B9 A0 0F F3 AA C3          秒2ノ2メマ!  ケ  Eェテ
```

コマンドメニュー3 (IN～JS)

IN (INput byte and INput word)
　アキュムレータへポートから入力
コード：オペコード表参照
フラグ：すべて変化せず
クロック数：オペコード表参照

IN 命令はI/Oポートからのデータをアキュムレータに入れます。BASICにもINP関数があるので分かると思います。8086のI/Oポートは0～255ではなく，0から65535まであります。

IN命令は

```
IN   AL, 37H
```

のように，I/Oポートアドレスが0～255の間はI/Oポートアドレスは直接指定して入力することができますが，255を超えるような場合，たとえばポートの1234Hから入力するような場合は

```
MOV     DX, 1234H
IN      AL, DX
```

のように**DXレジスタ間接**で入力します。PC-9801本体ではポートは0～255の間が使われているため，DXレジスタ間接でデータを入力することはありませんが，マウスのインタフェースでは7FDDH(Write),7FD9H(Read)のI/Oポートが使われているため，

```
MOV     DX, 7FD9H
IN      AL, DX
```

のようにDXレジスタ間接で読んでこなくてはなりません。BASICのINP関数は0～255のI/Oポートしか読めませんので注意してください。また，8086では

```
IN AX, 37H
```

のように2バイト一度に読んでくることが可能です。この場合

　AL レジスタにポート37Hの内容

AH レジスタにポート38Hの内容

が入ってきます。これは I/O ポートを介してワードデータのやりとりをするのには便利な命令ですが，ハードウェア自体が連続したポートにワードデータがくるようになっていなければ意味がありませんから，PC-9801では使うことはあまりないと思います。もちろん

IN　AX, DX

のように，**DXレジスタ間接**で読んでくることもできます。

また，入力に使えるレジスタはアキュムレータ，すなわち AL (バイト)，AX (ワード) だけであることにも注意してください。

INC (INCrement destination by 1)

インクリメント

コード：オペコード表参照

フラグ：AF, OF, PF, SF, ZF

クロック数：オペコード表参照

INC 命令はオペランドに 1 を足します。INC 命令のオペランドはオペコード表を見れば分かるとおり，**レジスタ，メモリの両方**が使えます。メモリを直接インクリメントできることは非常に助かります。セグメントレジスタに INC 命令を使うことはできません。セグメントレジスタに対しては，MOV と PUSH, POP 命令くらいしか使えないと考えてください。

INC 命令で重要なのは，これまでにも述べましたが，**キャリーフラグは変化しない**ことです。

```
MOV   AX, 0FFFFH
ADD   AX, 1
```

ではキャリーが立ちましたが，

```
MOV   AX, 0FFFFH
INC   AX
```

ではキャリーは変化しません。たとえば，

```
{INC    AX
{JB     LABEL
```

```
{ADD  AX, 1
{JB   LABEL
```
の例では AX が 0FFFFH のとき，違った動作になります。

INT（INTerrupt） 　ソフトウェアインタラプト コード：オペコード表参照 フラグ：IF，TF が影響を受ける クロック数：52

INT 命令はソフトウェアインタラプトを起こします。

8086ではアドレスの**00000H〜003FFH**に割り込みベクトルが書かれています。

```
INT  8
```
を実行して，インタラプトの8番をかけたとします。すると8086はフラグ，CS，IP の順にスタックに積み，アドレスの**20H〜23H** に書かれてるベクトルの指すアドレスに処理を移します。

ベクトルの内容は，

20H　21H	22H　23H
オフセット	セグメント

8番ベクトルは20Hからでしたが
```
INT 9  ……24H
INT 10 ……28H
```
のように
```
INT  N
```
の場合
```
N * 4
```
の位置にあるベクトルが参照されます。

PC-9801 の内部ルーチンを利用するには，この INT 命令を使います。内部ルーチンの一覧は各種解析書などにくわしく載っていますが，たとえば1文字

入力は

```
    MOV  AH, 00H
    INT  18H
      └→ALにASCIIコードが入る
```

のように行います。このようにROMの特定番地を直接コールするのでなく，内部割り込みを使うのはROMがバージョンアップされた場合でもユーザー側のプログラムをいっさい変更せずに動かすことができるためです。PC-9801，E，Fと変化する過程で，ROMの内容は少しずつ変化していますが，多くの機械語ソフトウェアが同シリーズで実行できるのもこのためといえます。

割り込みベクトルの0～4番は8086にとって特殊な意味があります。

0番　　0による除算

1番　　シングルステップ

2番　　NMI

　　　（ノンマスカブルインタラプト）

3番　　ブレークポイントの設定

4番　　オーバーフロー(次項参照)

たとえば，0で割り算が行われたときに，

```
    FD80H    :   082BH
    セグメント   オフセット
```

のルーチンに飛びたければ，

```
    00000 — 2B  08  80  FD
```

とすればよいのです。

シングルステップとは，8086が1命令を実行するたびにインタラプトをかけることで，デバッグのときに役立ちます。CP/M-86のDDT86のT（トレース）コマンドはこれを使っています。シングルステップを行うかどうかは，TF（トラップフラグ）を立てるかどうかによります。ユーザープログラムでシングルステップを使うことは，まずない（デバッガを作っているのなら話は別ですが）でしょう。

NMI（ノンマスカブルインタラプト）はハードウェア割り込みの1つですが，通常のハードウェア割り込みがマスクできるのに対し，NMIは文字どおりマスクすることはできません。NMIはハードウェアに重大な障害があると

きなどにかかるようにするのが普通です。PC-9801では，メモリのパリティエラーが生じたときにこの割り込みがかかるようになっています。NMIが実行されると，PC-9801では

MEMORY ERROR 1 （または2）

を表示するルーチンへ飛び，実行停止状態になります（1か2かは標準メモリのエラーか，拡張メモリのエラーかによります)。

話を元に戻しましょう。PC-9801ではベクトルの0～3FHがシステム側で予約されています。

ベクトル番号　40H～7FH

はユーザー用として自由に使える状態にあります。

またオペコード表を見れば分かりますが，INT命令は通常，

CDH　N　　　（Nはベクトル番号）

なる2バイト命令ですが，3番は

CCH

と1バイト命令となります。

この割り込みはブレークポイントの設定に使われます。1バイト命令ですから，ほかの8086のどんな命令とも入れ換えることができます。

INTO (INTerrupt if Overflow)
オーバーフロー時インタラプト
コード：CEH
フラグ：変化せず
クロック数：52

INTO命令はオーバーフローが生じたとき，つまりOFが1ならばタイプ4のインタラプトを発生させます。タイプ4ですから

$4\times4=16(=10H)$

つまり，10H～13Hのベクトルの指すルーチンに処理を移します。

INTOはオーバーフローエラーが起きた場合オーバーフローエラー処理ルーチンに飛ばしたいときに使います。が，数値演算のプログラムでない限り使うことはあまりないと思います。これも8086でコンパイラを書くのに都合がよい

という理由で設けられた命令でしょう。

> **IRET (Interrupt RETurn)**
> 　復帰
> コード：11001111=CFH
> フラグ：すべて変化
> クロック数：24

IRET 命令はインタラプト処理ルーチンから返るのに使います。

INT 命令で説明したように，インタラプトがかかるときには

　FLAG, CS, IP

がスタックに積まれています。これらを元に戻せば，呼んだ位置に返れるわけです。ですから，IRET 命令はスタックから

　IP, CS, FLAG

の順にポップします。スタックに何がいくつ積まれているかは，そのルーチンの呼ばれ方によります。

　インタラプトによるか
　セグメント内コールか
　セグメント間コールか

これらに応じて，復帰には，

　IRET, RET, RETF

を使い分けなければなりません。さもなければ，スタック上のデータが誤ったレジスタに戻されてしまいます。

インタラプト処理ルーチンからの復帰は間違いなく IRET 命令を使うようにしましょう。

> **JA (Jump if Above)**
> **JNBE (Jump if Not Below nor Equal)**
> 　>ならジャンプ
> 　≦でなければジャンプ
> コード：01110111=77H

フラグ：変化せず
クロック数：8（ジャンプした場合）
　　　　　　4（ジャンプしない場合）

JA，JNBE は＞の場合ジャンプします。JA，JNBE はオブジェクトコードは同一で，呼び方が違うにすぎません。

条件分岐命令は，CMP 命令の直後に使うことがほとんどです。

JA，JNBE 命令は符号なしデータの比較に使われます。8086の命令中に above，below とあれば必ず符号なしで，greater，less とあれば必ず符号つきです。たとえば，

```
JNLE
```

は Jump if Not Less nor Equal ですから符号つきデータだということが分かります。

JA，JNBE 命令では符号なしデータで＞のときにジャンプします。

```
MOV   AX, 5
CMP   AX, 3
JA    LABEL
```

では，

5 ＞ 3

ですから LABEL にジャンプします。

```
CMP   AX, 3
```

を実行したときにフラグがどうなっているか，ユーザーは関知する必要はありませんが

CF＝0，ZF＝0

となっています。もし AX が 3 であれば

CF＝0，ZF＝1

となりますし，AX が 2 であれば

CF＝1，ZF＝0

となります。つまり，＞のときに JMP を実行するためには，

CF＝0，ZF＝0

の場合に限りジャンプすればよいのです。

このJA, JNBE命令, 実は

CF=0, ZF=0

ならジャンプする命令なのです。しかし，プログラマがフラグの状態に従ってジャンプするかどうかを考えるのではあまりに原始的です。ですから,

JA, JNBE

というように比較的分かりやすい名前になっているといえます。もしフラグの変化によって名前をつけるとすれば,

JNCNZ

となるでしょうが，これではプログラマも大変です。

また，条件分岐命令はすべて−128〜127バイトの範囲しか飛べないことも重要です。

条件分岐命令は

コード	disp
	8ビット

となっていますが，分岐する際はdispが16ビットに符号拡張されてIPに加えられます。つまり,

(IP)←(IP)+disp (符号拡張)

という動作をします。このようなジャンプを相対ジャンプと呼びますが，dispが8ビットなので−128〜127の間しか飛べないわけです。−128〜127以上飛びたい場合は,「いったん−127〜126以内のラベルに飛んでおいて，そこからJMP命令で飛び直します。たとえば,「>だったら遠くのラベルLABELに飛ぶ」には,

```
        JA    TEMP
          ⌇
  TEMP：JMP  LABEL
```

のようにします。

ハンドアセンブルする際にはdispの値を計算するのは大変です。私自身，8ビットCPUではハンドアセンブルしたこともありますが，8086ではほとんどしたことがありません。やはりアセンブラにまかせるほうが無難ではないでしょうか。ハンドアセンブルは機械的な変換作業でプログラミングとはまったく関係ない作業ですから，やはり機械に行ってもらうほうが間違いもなく速く

行えます。特に16ビットCPUのプログラムは大きくなりがちですから，なおさらです。

JAE（Jump if Above or Equal）
JNB（Jump if Not Below）
　≧ならばジャンプ
　＜でなければジャンプ
コード：01110011＝73H
フラグ：変化せず
クロック数：16（ジャンプする場合）
　　　　　　4（ジャンプしない場合）

JAE，JNB は符号なしで≧のときジャンプする命令です。above，below が使われていますから，**符号なし**であることが分かります。フラグからいえば

　CF＝1……ジャンプしない
　CF＝0……ジャンプする

という動作をします。注意することは，前項と同様

- 符号なし
- －128～127しか飛べない
- CMP 命令の直後に使う（ことが多い）

点でしょう。いい忘れましたが－128～127しか飛べないにもかかわらず，それ以上飛ぼうとするとアセンブラが，

　LABEL OUT OF RANGE

のエラーメッセージで知らせてくれます。

JB（Jump if Below）
JNAE（Jump if Not Above nor Equal）
　＜ならばジャンプ
　≧でなければジャンプ
コード：01110010＝72H
フラグ：変化せず

クロック数：16（ジャンプするとき）
　　　　　　4（ジャンプしないとき）

JB, JNAE 命令は符号なしで＞のときにジャンプする命令です。below, above が使われていますから，**符号なし**であることが分かります。フラグからいえば

　CF＝1

のときにジャンプします。

JBE（Jump if Below or Equal）

JNA（Jump if Not Above）

　≦ならばジャンプ

　＞でなければジャンプ

コード：01110110＝76H

フラグ：変化せず

クロック数：16（ジャンプする場合）
　　　　　　4（ジャンプしない場合）

JBE, JNA 命令は符号なしで≦のときジャンプします。below や above が使われていますから**符号なし**です。フラグが

　CF＝1 OR ZF＝1

ならばジャンプします。

JCXZ（Jump if CX is Zero）

　CX＝0ならばジャンプ

コード：11100011＝E3H

フラグ：変化せず

クロック数：9（ジャンプする場合）
　　　　　　5（ジャンプしない場合）

JCXZ 命令は CX レジスタが0ならばジャンプする命令です。CX レジスタ

はカウンタとも呼ばれ，ループカウンタによく使われます。CXをカウントダウンしてCXが0ならば何かを行う，といった場合にこの命令は使えます。たとえば1～100を加えて，AXレジスタに入れるプログラムは

```
        MOV   AX, 0
        MOV   CX, 100
LABEL :
        ADD   AX, CX
        DEC   CX
        JCXZ  BREAK
        JMP   LABEL
BREAK :
```

と書けます。[注18]

> **JE（Jump if Equal）**
> **JZ（Jump if Zero）**
> 　＝ならジャンプ
> 　0ならジャンプ
> コード：01110100＝74H
> フラグ：変化せず
> クロック数：16（ジャンプするとき）
> 　　　　　　4（ジャンプしないとき）

JE, JZ命令は＝の場合ジャンプします。＝の判定は符号つきも符号なしも関係なく行えるので，符号つき，なしともにJE, JZ命令は使えます。フラグから見ると，ZFが1ならジャンプします。

> **JG（Jump if Greater）**
> **JNLE（Jump if Not Less nor Equal）**
> 　＞ならばジャンプ
> 　≦でなければジャンプ
> コード：01111111＝7FH

フラグ：変化せず
クロック数：16（ジャンプするとき）
　　　　　　4（ジャンプしないとき）

JG, JNLE 命令は符号つきで>のときジャンプします。greater や less が使われていますから符号つきであることが分かります。フラグを見ると複雑ですが，

(ZF＝0) AND (SF＝OF)

のときジャンプします。符号つきの場合，サインフラグとオーバーフローフラグがからんでくるのでやっかいですが，プログラマは，いちいちそんなことは考える必要はありません。「符号つきの>だから JG だな」と判断すればよいのです。

JGE (Jump if Greater Equal)
JNL (Jump if Not Less)
≧ならばジャンプ
<でなければジャンプ
コード：01111101＝7DH
フラグ：変化せず
クロック数：16（ジャンプするとき）
　　　　　　4（ジャンプしないとき）

JGE, JNL 命令は符号つきで≧ならばジャンプする命令です。greater や less が使われていますから**符号つき**です。

フラグを見ると

SF＝OF

ならジャンプとなります。

JL (Jump if Less)
JGNE (Jump if Greater Nor Equal)
<ならばジャンプ

≧でなければジャンプ
コード：01111100＝7CH
フラグ：変化せず
クロック数：16（ジャンプするとき）
4（ジャンプしないとき）

JL，JGNE 命令は符号つきで＜ならばジャンプする命令です。less や greater が使われているため，**符号つき**であることが分かります。

フラグが

SF≠OF

ならジャンプとなります。

JLE（Jump if Less or Equal）
JNG（Jump if Not Greater）
≦ならばジャンプ
＞でなければジャンプ
コード：01111110＝7EH
フラグ：変化せず
クロック数：16（ジャンプするとき）
4（ジャンプしないとき）

JLE，JNG 命令は符号つきで≦のときにジャンプする命令です。less や greater が使われていますから，**符号つき**です。

フラグが

(ZF＝1) OR (SF≠OF)

ならジャンプとなります。

JMP（Jump）
ジャンプ
コード：オペコード表参照
フラグ：変化せず

クロック数：オペコード表参照

JMP 命令はオペランドにジャンプします。JMP 命令には，表2-1の5つがあります。それぞれのオブジェクトコードは違いますが，アセンブラがオペランドのタイプなどから自動的に判断してくれるため，プログラマは関知する必要がありません。

表2-1

JMP	セグメント内	直接ショートジャンプ	JMP SHORT NEAR LABEL
		直接ジャンプ	JMP NEAR LABEL
		間接ジャンプ	JMP WORD PTR [BX] [SI]
	セグメント外	直接ジャンプ	JMP FAR LABEL
		間接ジャンプ	JMP DWORD PTR [BX] [SI]

JNE（Jump if Not Equal）

JNZ（Jump if Not Zero）

キならばジャンプ

0でなければジャンプ

コード：01111011＝7BH

フラグ：変化せず

クロック数：16（ジャンプするとき）

4（ジャンプしないとき）

JNE, JNZ 命令はキでなければジャンプする命令です。符号つき，なし両方に使えます。フラグが

ZF＝0

ならばジャンプします。

JNO（Jump if Not Overflow）

オーバーフローでなければジャンプ

コード：01110001＝71H

フラグ：変化せず

クロック数：16（ジャンプするとき）

4 (ジャンプしないとき)

JNO 命令では，オーバーフローでなければジャンプします。つまり，

OF[注19] = 0

ならばジャンプします。

JNP (Jump if Not Parity)

JPO (Jump if Parity Odd)

パリティフラグが0ならばジャンプ

パリティが奇数ならばジャンプ

コード：01111011＝7BH

フラグ：変化せず

クロック数：16 (ジャンプするとき)

4 (ジャンプしないとき)

JNP, JPO 命令はパリティが奇数ならばジャンプします。

PF[注20] = 0

ならばジャンプするといってもよいでしょう。

JNS (Jump if Not Sign)

サインフラグが0ならジャンプ

コード：01111001＝79H

フラグ：変化せず

クロック数：16 (ジャンプするとき)

4 (ジャンプしないとき)

JNS 命令はサインフラグ[注21]が0ならばジャンプします。

JO (Jump if Overflow)

オーバーフローならジャンプ

コード：01110000＝70H

フラグ：変化せず
クロック数：16（ジャンプするとき）
　　　　　　4（ジャンプしないとき）

JO 命令は OF＝1のときジャンプします。

JP（Jump if Parity）
JPE（Jump if Parity Even）
　パリティフラグが1ならジャンプ
　パリティが偶数ならばジャンプ
コード：01111010＝7AH
フラグ：変化せず
クロック数：16（ジャンプするとき）
　　　　　　4（ジャンプしないとき）

JP，JPE 命令は PF が1ならばジャンプします。

JS（Jump if Sign）
　サインフラグが1ならばジャンプ
コード：01111000＝78H
フラグ：変化なし
クロック数：16（ジャンプするとき）
　　　　　　4（ジャンプしないとき）

JS 命令は SF が1ならばジャンプする命令です。
ジャンプ命令がたくさん出てきて，すべてを覚えるのは大変そうですが，次の表で覚えてください。

	符号つき	符号なし
＞	JG	JA
≧	JGE	JAE

=	JE	JE
≦	JLE	JBE
<	JL	JB

数値の比較ではこれだけしかありません。あとは符号つきかなしかを考えれば間違いなくプログラミングできるでしょう。

greater, less ……符号つき

above, below……符号なし

もお忘れなく。

CP/M-86によるプログラミング

IN 命令から JS 命令までを説明しました。次に簡単なアセンブル例を載せておきます。これは1～100の平方数を求めるプログラムです。BASICでは

```
100  FOR I=1 TO 100
110   FOR J=1 TO I
120    FOR K=1 TO J
130     IF I*I=J*J+K*K  IF  THEN PRINT I;"*";I;
     "=";J;"*";J;"+";K;"*";K
140    NEXT K
150   NEXT J
160  NEXT I
170  END
```

とでもなるでしょうか。これを CP/M-86 の上で書いてみました（リスト2-12)。実行して，速さを BASIC と比べてください。

まず，リストをファイル名 OHPC8. A86 で打ち込み，

```
ASM86  OHPC8  $PZ  SZ  ⏎
GENCMD  OHPC8  8080  ⏎
```

とします。実行は

```
OHPC8  ⏎
```

です。

リスト2-12

```
;
; Oh! PC DEMO PROGRAM
;
        CSEG
        ORG     100H
START:
        MOV     AX,CS           ! MOV   DS,AX  8080モデル

        MOV     I,1
FOR_I:
        MOV     J,1
FOR_J:
        MOV     K,1
FOR_K:
        MOV     AX,J            ! IMUL  AX⎫ BX=J*J
        MOV     BX,AX                     ⎭

        MOV     AX,K            ! IMUL  AX⎫ BX=J*J+K*K
        ADD     BX,AX                     ⎭

        MOV     AX,I            ! IMUL  AX  AX=I*I

        CMP     AX,BX        ⎫
        JNE     PYTAG1       ⎬ I*I=J*J+K*Kならば画面出力
                             ⎪
        CALL    PRINT_I_J_K  ⎭
PYTAG1:
        INC     K      ⎫
        MOV     AX,K   ⎪
        CMP     AX,J   ⎬ NEXT K
        JLE     FOR_K  ⎭

        INC     J      ⎫
        MOV     AX,J   ⎪
        CMP     AX,I   ⎬ NEXT J
        JLE     FOR_J  ⎭

        INC     I      ⎫
        CMP     I,100  ⎬ NEXT I
        JLE     FOR_I  ⎭

        XOR     DL,DL  ⎫
        XOR     CL,CL  ⎬ CP/M-86へ
        INT     224    ⎭

PRINT_I_J_K:
        MOV     AX,I    ! CALL  PRINT_AX ⎫
        MOV     AX,'*'  ! CALL  PRINT    ⎬ I*Iを表示
        MOV     AX,I    ! CALL  PRINT_AX ⎭
        MOV     AL,'='  ! CALL  PRINT      "="を表示
        MOV     AX,J    ! CALL  PRINT_AX ⎫
        MOV     AX,'*'  ! CALL  PRINT    ⎬ J*Jを表示
        MOV     AX,J    ! CALL  PRINT_AX ⎭

        MOV     AL,'+'  ! CALL  PRINT      "+"を表示

        MOV     AX,K    ! CALL  PRINT_AX ⎫
        MOV     AL,'*'  ! CALL  PRINT    ⎬ K*Kを表示
        MOV     AX,K    ! CALL  PRINT_AX ⎭
        MOV     AL,0DH  ! CALL  PRINT    ⎫ CR, LF
        MOV     AL,0AH  ! CALL  PRINT    ⎭
        RET
PRINT:
        MOV     CL,2    ! MOV   DL,AL    ⎫ CP/M-86のBDOSコールを
        INT     224                      ⎬ 使ってAレジスタの
        RET                              ⎭ コードを画面へ出力
```

```
PRINT_AX:
        MOV     CX,5      } AXレジスタの値を10進出力
        MOV     BL,10
PRINT1:
        IDIV    BL
        PUSH    AX
        CBW               } 上位の桁から各桁をスタックにプッシュ
        LOOP    PRINT1
        MOV     PRESS,-1
        MOV     CX,4
PRINT2:
        POP     AX
        MOV     AL,AH
        CMP     PRESS,-1          ! JNE    PRINT5

        CMP     AL,0     ! JE     PRINT3    先頭の0は印字しない

        MOV     PRESS,0
PRINT5:
        ADD     AL,'0'
        PUSH    CX        } 0以外は印字
        CALL    PRINT
        POP     CX
PRINT3:
        LOOP    PRINT2
        POP     AX
        MOV     AL,AH    ! ADD    AL,'0'    最後の1桁は無条件に印字
        CALL    PRINT
        RET
END_CS:
        DSEG                 } 8080モデル
        ORG     OFFSET END_CS
I       DW      1
J       DW      1            } 変数エリア
K       DW      1
PRESS   DW      -1
        END
```

コマンドメニュー 4 (LAHF～OUT)

LAHF (Load AH from Flags)

フラグを AH レジスタにロード

コード：10011111＝9FH

フラグ：変化せず

クロック数： 4

LAHF 命令はフラグの状態を AH レジスタにロードします。8086のフラグは

O D I T S Z A P C

となっていますが，LAHF 命令で AH レジスタにロードされるフラグは

SZ * AF * PF * CF　　（*は不定）

の順になります。これは8080のフラグの並びと同じです。8080のエミュレーションには便利かもしれません。

LDS (Load Data Segment register)

レジスタと DS にロード

コード：オペコード表参照

フラグ：変化せず

クロック数：オペコード表参照

LDS 命令はレジスタと DS レジスタにデータをセットします。高級言語などでは，変数の管理をその変数の格納されているアドレスのオフセットとセグメントをテーブルにもっていきます。たとえば，整数変数Xのオフセットとセグメントが POINTER_X に入っているとします。すると，Xの値を AX レジスタにロードするには，

```
LDS   SI, POINTER_X
MOV   AX, [SI]
```

と，非常に簡潔に表せます。このように，8086はコンパイラを作るのに都合のよい命令を数多くもっています（8086は PL/M を強く意識して作られています）。

LDS 命令で大切なのは，ロードされる順番でしょうか。レジスタにロードされ，次に DS レジスタにロードされます。メモリには，

レジスタ値，DS レジスタ値

の順番に格納されている必要があります。8086では，割り込みベクトルを見れば分かるように，オフセット，セグメントの順に格納することが多いのですが，LDS 命令でも同じことがいえます。これは普通の思考の逆のようですが，16ビットデータの格納も

LOW　HIGH

の順になる80系のことですから，

オフセット　セグメント

の順のほうが自然なのかもしれません。

LEA（Load Effective Address）

オフセットアドレスのロード

コード：オペコード表参照

フラグ：変化せず

クロック数：オペコード表参照

LEA 命令はソースのオフセットをレジスタにロードします。変数Xのオフセットを SI にロードするには

```
LEA   SI, X
```

とします。アセンブラには OFFSET という演算子があって，いまの例は

```
MOV   SI, OFFSET X
```

と書くこともできます。しかし，オフセットが実行時に変化する場合は OFFSET 演算子は使えません。たとえば，ARRAY_X [SI] のオフセットを SI にロードするのに

```
MOV   SI,
          OFFSET ARRAY_X [SI]
```

とはできません。

LEA　SI, ARRAY_X [SI]

と LEA 命令を使うしかないのです。

LES (Load Extra-Segment register)

レジスタと ES にロード

コード：オペコード表参照

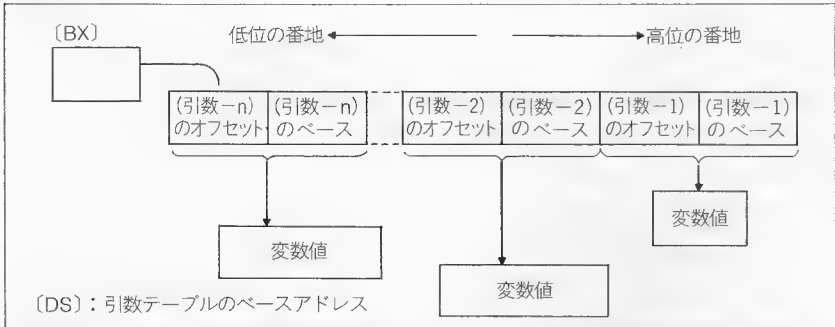
フラグ：変化せず

クロック数：オペコード表参照

LES 命令はレジスタと ES レジスタの両方を一度にセットします。DS が ES になっただけで，ほかは LDS 命令と同じです。

LES 命令は BASIC から機械語ルーチンで引数を受けるときに非常に便利です。

PC-9801 のマニュアルを見れば分かるとおり，CALL 文の引数は図2-2のようになっています。つまり機械語ルーチンに制御が渡されたときに DS, BX レジスタに引数テーブルのセグメント，オフセット値がセットされており，引数テーブルには引数のオフセットとセグメント値が書かれているわけです。

図2-2

引数が整数変数の例を考えてみましょう。その変数の値を AX レジスタにセットするには，

LES　SI, [BX]

```
    MOV   AX, ES : [SI]
```
ですみます。もしLES命令がなかったら
```
    MOV   SI, [BX]
    MOV   ES, 2[BX]
    MOV   AX, ES : [SI]
```
とするところでしょう。

いまの引数が1つの場合には,
```
    LDS   SI, [BX]
    MOV   AX, [SI]
```
とLDS命令で置き換えることもできますが,引数が2つ以上の場合LDS命令は不便です。第1引数をCXに第2引数をDXにロードすることを考えましょう。LESでは
```
    LES   SI, 4[BX]
    MOV   CX, ES : [SI]
    LES   SI, [BX]
    MOV   DX, ES : [SI]
```
となりますが,ここではLDSでは置き換えられません。
```
    LDS   SI, 4[BX]
    MOV   CX, [SI]
    LDS   SI, [BX]
    MOV   DX, [SI]
```
では第1引数はセットされますが,第2引数はうまく渡されません。DSレジスタが最初のLDS命令で変更されたためです。

LOCK (LOCK)

ロック信号の設定

コード:11110000=F0H

フラグ:変化せず

クロック数: 2

LOCK命令は,一種のプレフィックスで,後続する命令の実行中に8086の

$\overline{\mathrm{LOCK}}$ 信号をローにします。これはマルチプロセッサシステムでの共有メモリのアクセスに役に立ちます。

たとえば，共有メモリに CTRL があるとします。

LOCK　XCHG AL，CTRL

とすれば，この8086が，

XCHG　AL,CTRL

を行っている間，ほかの CPU によって CTRL の値が置き換えられることはありません。ただし，それはハードウェアが8086の $\overline{\mathrm{LOCK}}$ 信号によってバスの優先権を判定する構成になっている場合に限られます。PC–9801 で LOCK 命令を使うことはないと思います。

LODS (LOaD byte or word String)

AL/AX レジスタへのロード

コード：オペコード表参照

フラグ：変化しない

クロック数：12

LODS 命令は AL/AX レジスタにデータをロードします。AL レジスタにロードする場合は

LODSB

AX レジスタにロードする場合は

LODSW

と書きます。データは

DS：[SI]

から AL/AX レジスタにロードされます。このとき重要なのは，ディレクションフラグ DF により，SI レジスタが自動的にインクリメントかデクリメントされることです。つまり，

DF＝0　→ SI＝SI＋1 or 2

DF＝1　→ SI＝SI－1 or 2

1増減，2増減されるかは LODSB，LODSW によります。これは非常に便利です。

たとえば文字列のプリントを考えます。1文字出力ルーチンが，ONCOUT であったとし，文字列が

```
STRING   DB
         `THIS IS STRING.′, 0
```

となっていたとします。プログラムは，

```
        MOV   SI, OFFSET  STRING
        CLD
   L1 :
        LODSB
        CMP   AL, 0
        JE    L2
        CALL  ONCOUT
        JMP   L1
   L2 :
```

と，とても簡単になります。LODS 命令で

```
   MOV   AL, [SI]
   INC   SI
```

の働きを実現しているわけです。

LODS 命令で重要なのは，ディレクションフラグのセットを忘れないことです。DF の値によって，SI は勝手にインクリメント/デクリメントをしてしまいますから。

LOOP（LOOP）

CX をデクリメントし，非0でジャンプ

コード：11100010＝E2H

フラグ：変化せず

クロック数：17（ジャンプするとき）

5（ジャンプしないとき）

LOOP 命令は CX レジスタを1デクリメントしても，0でなければジャンプします。たとえば，1～100の和は

```
        MOV   AX, 0
        MOV   CX, 100
    L1: ADD   AX, CX
        LOOP  L1
```

で簡単に（AX レジスタ）に求められます。LOOP 命令は

```
    DEC   CX, 1
    CMP   CX, 0
    JNE   LABEL
```

と同じような働きがあるといえます。

LOOP 命令で注意することは，ジャンプできる範囲が

−128〜127

の間である点です。もしこれ以上のジャンプをしようとすれば，アセンブラが

LABEL OUT OF RANGE

と警告してくれます。そのような場合

```
    LOOP  LABEL
```

の代わりに

```
            LOOP  TEMP
            JMP   NEXT
    TEMP:   JMP   LABEL
    NEXT:
```

のように一度近くに飛んで，そこからあらためてジャンプし直せばよいでしょう。

LOOPZ/LOOPE(LOOP if Zero/LOOP if Equal)

CX をデクリメント，

CX≠0, ZF=1 ならジャンプ

コード：11100001=E1H

フラグ：変化せず

クロック数：18（ジャンプするとき）

5（ジャンプしないとき）

LOOPZ/LOOPE 命令は CX レジスタをデクリメントし，

CX≠0，ZF＝1

ならばジャンプします。

たとえば，配列のデータで最初に`A′でないものの位置を調べるには

```
        MOV   SI, −1
        MOV   CX, LENGTH_OF_ARRAY
    L1：
        INC    SI
        CMP   ARRAY [SI],`A′
        LOOPZ  L1
        JNZ    ERROR   全部`A′だった
         ：(SI に場所が入っている)
    ERROR： 全部`A′だった
```

となります。LOOPZ/LOOPE で大切なのは，フラグが変化しないことです。いまの例でも

CMP ARRAY [SI],`A′

で変化したフラグが，LOOPZ で変化しないために，

JNZ ERROR

で条件ジャンプ（全部`A′だった）できるわけです。

LOOPNZ/LOOPNE（LOOP if Not Zero/LOOP if Not Equal）

CX をデクリメント

CX≠0，ZF＝0 ならジャンプ

コード：11100000＝E0H

フラグ：変化せず

クロック数：19（ジャンプするとき）

5（ジャンプしないとき）

LOOPNZ/LOOPNE 命令は CX レジスタをデクリメントし，

CX≠0，ZF＝0

ならばジャンプします。

これは次のようなときに使えます。
「ストリング中の 'A' の位置を調べる」プログラムは

```
      MOV   SI, -1
      MOV   CX, LENGTH_OF_STRING
  L1:
      INC   SI
      CMP   STRING [SI],'A'
      LOOPNE  L1
      JNE   ERROR ('A'がない)
            : SI に 'A' の位置
  ERROR:
```

LOOPNZ/LOOPNE 命令もフラグを変化させないことが重要です。

MOV (MOVe)

転送
コード：オペコード表参照
フラグ：変化せず
クロック数：オペコード表参照

MOV 命令はデータの転送を行います。MOV で重要なのは，何から何への転送ができるか，つまりオペランドには何が選べるかです。大別すると

レジスタ⇆レジスタ
レジスタ⇆メモリ
レジスタ←イミディエイト
メモリ←イミディエイト
セグメントレジスタ⇆レジスタ
セグメントレジスタ⇆メモリ

となるでしょうか。初めて8086のアセンブリ言語を使って間違えてしまうのは

```
MOV   DS, 0A800H
```

というふうにセグメントレジスタにイミディエイトデータを入れることです。上の表を見れば分かるとおり，

「セグメントレジスタにイミディエイト値は入れられない」
のです。セグメントレジスタには，レジスタまたはメモリからロードしなくてはなりません。たとえば，CS と DS を一致させるには，

```
MOV   AX, CS
MOV   DS, AX
```

とするか

```
PUSH  CS
POP   DS
```

とするしかありません。

```
MOV   DS, CS
```

とはできないのです。

一方，メモリにイミディエイトデータが転送できるのはきわめて便利です。たとえば，変数Xに1234を代入するには，単に

```
MOV   X, 1234
```

とすればよいのです。

しかし，メモリからメモリへの転送はできません。XにYを代入するには

```
MOV   AX, Y
MOV   Y, AX
```

とでもするしかありません。

```
MOV   X, Y
```

とはできないのです。

何と何の間で転送できるかは使って覚えるしかないでしょう。間違っていれば，アセンブラが，

```
OPERAND(S) MISMATCH INSTRUCTION
```

と知らせてくれますから。

MOVS (MOVe String)

メモリ間転送

コード：オペコード表参照

フラグ：変化せず

クロック数：18（一度だけのとき）

9+(17*R)　REP プレフィックスでR回実行したとき

MOVS 命令はメモリ間転送を行います。転送は

DS：[SI] → ES：[DI]

です。このようにソースのデフォルトセグメントはDSですが，セグメントオーバーライトプレフィックスで変更できます。デスティネーションのセグメントは変更できません。

MOVS 命令も LODS 命令と同様，転送後に SI，DI レジスタがインクリメント/デクリメントされます（DF フラグによる）。

バイト転送の場合は

MOVSB

ワード転送の場合

MOVSW

を使います。

MOVS 命令は単独で使われることはまれで，たいてい

REP プレフィックス

を前置きして使います。REP プレフィックスは，後続するストリング命令を CX がデクリメントされて 0 になるまで繰り返すことを指定します。

MOVS 命令は明らかにブロック転送に向いています。たとえば，PC-9801 で青の VRAM のデータを赤の VRAM に転送するには

```
CLD
MOV   AX, 0A800H  }
MOV   DS, AX      } 青の VRAM は A8000H から
MOV   AX, 0B000H  }
MOV   ES, AX      } 赤の VRAM は B0000H から
MOV   SI, 0
MOV   DI, 0
MOV   CX, 4000H   4000H ワード転送
REP   MOVSW
```

で行えます。

これはストリング命令すべてにいえることですが，

ソース→　　　　　　DS：[SI]

デスティネーション→ES：[DI]

となっています。SI はソースインデックスレジスタ，DI はデスティネーションインデックスレジスタですから，この使い方は納得のいくものでしょう。ストリング命令以外の場合も，ソースなら SI，デスネーションなら DI と使い分けるほうが無難です。つまり，

MOV　[SI]，123

はもちろん正しいのですが，もしレジスタに何を使ってもよいのならば，

MOV　[DI]，123

としたほうがよいと思います。

MUL（MULtiply）

乗算

コード：オペコード表参照

フラグ：CF，OF 変化

クロック数：オペコード表参照

（かなりかかる）

MUL 命令は**符号なし**データの乗算を用います。MUL 命令ではかけられる数は

AL（8ビットの場合）

AX（16ビットの場合）

で，かける数はレジスタかメモリのデータであり，結果は，

AX（8ビットの場合）

DX：AX（16ビットの場合）

となります。

MUL 命令は符号なしデータの乗算命令ですから，符号つきデータの乗算には使えません（IMUL 命令で行います）。

また，イミディエイト値の乗算もできません。

MUL　3

とはできなくて

```
MOV   BX, 3
MUL   BX
```

としなくてはなりません。

また，乗算命令は非常に時間がかかることも重要です。速度が要求される場合はシフトで置き換えられるならば，そのようにしたほうがよいでしょう。たとえば，AX を3倍するのならば，

```
MOV   BX, 3
MUL   BX
```

でもよいのですが，

```
MOV   BX, AX
SHL   AX, 1
ADD   AX, BX
```

としたほうがはるかに速くなります。

乗除算命令は16ビットCPUであることの証明のようなものですが，8086の乗除算命令は68000，Z8000などに比べると，かなり遅いといえます。この点はよく覚えておいてください。

NEG（NEGate）

2の補数をとる

コード：オペコード表参照

フラグ：AF，CF，OF，PF，SF，ZF が変化

クロック数：オペコード表参照

NEG 命令はオペランドの符号を反転します。たとえば，

```
MOV   AL, 2
NEG   AL
```

ならば，AL には－2が入っています。2の補数をとるというのは，

①ビットごとの反転をとる

②1を加える

ことですが，いまの②でこれを行うと，

2＝00000010

①11111101

②11111110＝－2

つまり，２の補数をとることと，符号を反転することは同値です。

NOP（No OPeration）

ノーオペレーション

コード：10010000＝90H

フラグ：変化しない

クロック数：３

NOP 命令は何も行いまん。

MS-DOS のマクロアセンブラには

EVEN

という擬似命令があります。これは偶数アドレスにロケーションカウンタを合わせる命令ですが，このとき奇数アドレスにロケーションがあった場合，

NOP

が１つ入ります。

NOT（NOT）

反転

コード：オペコード表参照

フラグ：変化しない

クロック数：オペコード表参照

NOT 命令はビットごとの反転（１の補数）をとります。

```
MOV   AL，11001100B
NOT   AL
```

ならば，

AL＝00110011B

となっています。

NOT で重要なのは，メモリを直接反転できることでしょう。8086では，多

くの命令でオペランドにメモリが指定でき，プログラミングが楽になっています。

次の命令はいずれも正しいものです。

NOT　(OFFSET ARRAY_X) [SI] [BX]

NOT　ARG 1 [BP]

NOT　X

OR (OR)

オア

コード：オペコード表参照

フラグ：CF，OF，PF，SF，ZF 変化　AF 不定

クロック数：オペコード表参照

OR 命令はビットごとのオア（論理和）をとります。BASIC にも OR はあるので，よく分かっていると思います。たとえば，

MOV　AL，10101010B

OR　　AL，01010101B

ならば，

AL＝11111111B

となります。

オペランドには

デスティネーション	ソース
レジスタ	イミディエイト
レジスタ	メモリ
レジスタ	レジスタ
メモリ	イミディエイト
メモリ	レジスタ

が指定できます。ですから，

OR　DATA, 1

OR　WORD PTR (OFFSET ARRAY) [SI] [BX], 3

などは正しい命令です。

OUT (OUTput)

アウト

コード：オペコード表参照

フラグ：変化しない

クロック数：オペコード表参照

OUT 命令は8086の I/O ポートにデータを出力します。IN 命令のところで述べたように，8086には

0 ～65535

の I/O ポートがあり，そのうち

0 ～255

が

OUT　37H，AL

のように直接ポートアドレスを指定できますが，256以上のポートを使う場合は

MOV　DX，256

OUT　DX，AL

のように，

DX レジスタ間接

でなければなりません。

また，ワード出力も可能で

OUT　37H，AX

とすると 37H に AL の内容が，38H に AH の内容が出力されます。もちろん

OUT DX，AX

のように DX レジスタ間接のときもこれは使えます。ただ，これは連続したポートアドレスにワード出力用のポートが割り振られている場合にしか使えませんので，PC-9801 本体だけの場合には，使うことはないと思います。

なお，OUT 命令で使えるのは

AL または AX レジスタ

のみです。

OUT　37H, BL

などとはできません。

プログラミングから実行まで

LAHF 命令から OUT 命令までを解説しました。アセンブリ言語のプログラミング例として，ランダムに点を打つ例をあげておきます。

これは BASIC では

```
FOR  I=1 TO 1000
     PSET (RND(1)*639, RND(1)*399), RND(1)*7
NEXT I
```

とでもなるでしょうか。

ここでは，MS-DOS，CP/M-86 両方のリストをあげ，チェックサムつきダンプリストもつけておきました。

CP/M-86

リスト2-13が CP/M-86 用ソースリストです。

エディタで打ち込み

>ASM86 OHPC9 $PZ SZ ⏎

でアセンブル，

>GENCMD OHPC9 8080 ⏎

で実行可能なファイルを作り，

>OHPC9

で実行します。

コメントは MS-DOS のソースリストを参照してください。

リスト2-13

```
;
; Oh PC
;
;        asm86 ohpc9 $pz sz
;        gencmd ohpc9 8080
```

```
;        ohpc9

TRUE     EQU      0FFFFH
FALSE    EQU      NOT TRUE
```

```
BLUE    EQU     0A800H
RED     EQU     0B000H
GREEN   EQU     0B800H
        CSEG
        ORG     100H
BEGIN:
        MOV     AX,CS
        MOV     DS,AX

        CALL    START
        CALL    ALL

        MOV     CX,1000
MAIN_LOOP:
        PUSH    CX
        MOV     AX,639+1
        CALL    RND
        MOV     X,AX
        MOV     AX,399+1
        CALL    RND
        MOV     Y,AX
        MOV     AX,7+1
        CALL    RND
        MOV     COLOR,AX

        CALL    WRITE_PIXEL

        POP     CX
        LOOP    MAIN_LOOP

        MOV     DL,0
        MOV     CL,0
        INT     224

START:
        MOV     AH,40H
        INT     18H
        RET
ALL:
        MOV     AH,42H
        MOV     CH,0C0H
        INT     18H
        RET

RND:
;
; ARGUMENT         AX:MAX_RANDOM
;       RETURN  AX:RANDOM NUMBER
;
        MOV     CX,AX              ; SAVE AX
        MOV     AX,259
        MUL     SEED
        ADD     AX,3
        AND     AX,32767
        MOV     SEED,AX
        MUL     CX
        MOV     BX,32767
        DIV     BX
        RET

 WRITE_PIXEL:
          MOV     PLOT,FALSE
          TEST    COLOR,1
          JZ      PLOT1
          MOV     PLOT,TRUE
 PLOT1:
          MOV     AX,BLUE
        CALL    PLOT_0
        MOV     PLOT,FALSE
        TEST    COLOR,2
        JZ      PLOT2
        MOV     PLOT,TRUE
PLOT2:
        MOV     AX,RED
        CALL    PLOT_0
        MOV     PLOT,FALSE
        TEST    COLOR,4
        JZ      PLOT3
        MOV     PLOT,TRUE
PLOT3:
        MOV     AX,GREEN
        CALL    PLOT_0
        RET

PLOT_0:
        MOV     ES,AX

        MOV     AX,X
        SHR     AX,1
        SHR     AX,1
        SHR     AX,1
        MOV     SI,AX

        MOV     AX,Y
        MOV     BX,AX
        SHL     AX,1
        SHL     AX,1
        ADD     AX,BX

        SHL     AX,1
        SHL     AX,1
        SHL     AX,1
        SHL     AX,1

        ADD     SI,AX
        MOV     AL,128
        MOV     CX,X
        AND     CL,7

        SHR     AL,CL
        CMP     PLOT,TRUE
        JNZ     XPLOT
        OR      ES:[SI],AL
        RET

XPLOT:
        NOT     AL
        AND     ES:[SI],AL
        RET

END_CS:
        DSEG
        ORG     OFFSET END_CS

X       DW      0
Y       DW      0
COLOR   DW      0
PLOT    DW      FALSE

SEED    DW      0

        END
```

MS-DOS

リスト2-14がMS-DOS用ソースリストです。

エディタで打ち込み,

　>MASM OHPC9 ; ⏎

でアセンブル,

　>LINK OHPC9 ; ⏎

で実行可能なファイルを作ります。実行は

　>OHPC9

です。

リスト2-14

```
;
; MS-DOS VER. 2. 0用
;
; 実行方法
; このリストを打ち込んでファイル名をOHPC9.ASMとし
; MASM OHPC9;
; LINK OHPC9;
; OHPC9
;
TRUE     EQU      0FFFFH            ┐
FALSE    EQU      NOT TRUE          │
                                    │ 定数の定義
BLUE     EQU      0A800H            │
RED      EQU      0B000H            │
GREEN    EQU      0B800H            ┘

STACK1   SEGMENT STACK              ┐
         DW       200 DUP (?)       │ スタックセグメント
TOP_OF_STACK      LABEL   WORD      │
STACK1   ENDS                       ┘

DATA     SEGMENT WORD               ┐
                                    │
X        DW       0                 │
Y        DW       0                 │
COLOR    DW       0                 │ データセグメント
PLOT     DW       FALSE             │
                                    │
SEED     DW       0                 │
DATA     ENDS                       ┘

CODE     SEGMENT WORD               以下コードセグメント
         ASSUME   CS:CODE,DS:DATA,SS:STACK1
BEGIN:
         CLI
         MOV      AX,STACK1         ┐
         MOV      SS,AX             │ SS, SPのセット
         MOV      SP,OFFSET TOP_OF_STACK ┘
         STI
```

```
        MOV     AX,DATA                          } DSのセット
        MOV     DS,AX

        CALL    START                            SCREEN, 0の指定
        CALL    ALL                              SCREEN, 3の指定

        MOV     CX,1000                          1000個の点を打つ
MAIN_LOOP:
        PUSH    CX
        MOV     AX,639+1
        CALL    RND                              X=RND(1)*639
        MOV     X,AX
        MOV     AX,399+1
        CALL    RND                              Y=RND(1)*399
        MOV     Y,AX
        MOV     AX,7+1
        CALL    RND                              COLOR=RND(1)*7
        MOV     COLOR,AX

        CALL    WRITE PIXEL                      点を打つ

        POP     CX
        LOOP    MAIN_LOOP                        本文LOOP命令参照

        MOV     AH,4CH
        MOV     AL,0                             } DOSに戻る
        INT     21H

START:
        MOV     AH,40H                           グラフィックVRAMの表示開始
        INT     18H
        RET
ALL:
        MOV     AH,42H
        MOV     CH,0C0H                          640×400モード
        INT     18H
        RET

RND:                                             乱数ルーチン
;
; ARGUMENT       AX:MAX RANDOM
;       RETURN  AX:RANDOM NUMBER
;
        MOV     CX,AX           ; SAVE AX
        MOV     AX,259
        MUL     SEED                             SEED=(259*SEED+3)AND32767
        ADD     AX,3
        AND     AX,32767
        MOV     SEED,AX
        MUL     CX
        MOV     BX,32767                         AX=SEED*AX 32767
        DIV     BX
        RET

WRITE_PIXEL:                                     点を打つルーチン
        MOV     PLOT,FALSE
        TEST    COLOR,1
        JZ      PLOT1
        MOV     PLOT,TRUE
PLOT1:
        MOV     AX,BLUE
        CALL    PLOT_0
        MOV     PLOT,FALSE
        TEST    COLOR,2
        JZ      PLOT2
        MOV     PLOT,TRUE
```

```
PLOT2:
        MOV     AX,RED
        CALL    PLOT_0
        MOV     PLOT,FALSE
        TEST    COLOR,4
        JZ      PLOT3
        MOV     PLOT,TRUE
PLOT3:
        MOV     AX,GREEN
        CALL    PLOT_0
        RET

PLOT_0:                                 AXレジスタで示されるセグメントのVRAMに
        MOV     ES,AX                   点を打つルーチン

        MOV     AX,X
        SHR     AX,1
        SHR     AX,1
        SHR     AX,1
        MOV     SI,AX

        MOV     AX,Y
        MOV     BX,AX
        SHL     AX,1                    AX=5*Y
        SHL     AX,1
        ADD     AX,BX

        SHL     AX,1
        SHL     AX,1                    AX=16*5*Y
        SHL     AX,1
        SHL     AX,1

        ADD     SI,AX                   SIにアドレスが入る
        MOV     AL,128
        MOV     CX,X                    CL=X MOD 8
        AND     CL,7

        SHR     AL,CL
        CMP     PLOT,TRUE
        JNZ     XPLOT
        OR      ES:[SI],AL              点を打つ
        RET

XPLOT:
        NOT     AL
        AND     ES:[SI],AL              点を消す
        RET

CODE    ENDS

        END     BEGIN                   beginから実行開始
```

モニタ

BASICモードから

MON ⏎

]C 1 F00 ⏎

でリスト2-15のダンプリストを打ち込みます。

リスト2-15

```
ADRS : +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +A +B +C +D +E +F :SUM
0000 : 50 53 51 52 56 57 55 1E 06 8C C8 8E D8 E8 32 00 : 40
0010 : E8 34 00 B9 E8 03 51 B8 80 02 E8 31 00 A3 F5 00 : FC
0020 : B8 90 01 E8 28 00 A3 F7 00 B8 03 00 E8 1F 00 A3 : 5D
0030 : F9 00 E8 33 00 59 E2 DE 07 1F 5D 5F 5E 5A 59 5B : 7B
0040 : 58 CF B4 40 CD 18 C3 B4 42 B5 C0 CD 18 C3 8B C8 : 29
0050 : B8 03 01 F7 26 FD 00 05 03 00 25 FF 7F A3 FD 00 : 21
0060 : F7 E1 BB FF 7F F7 F3 C3 C7 06 FB 00 00 00 F7 06 : 83
0070 : F9 00 01 00 74 06 C7 06 FB 00 FF FF B8 00 A8 E8 : 82
0080 : 35 00 C7 06 FB 00 00 00 F7 06 F9 00 02 00 74 06 : 6F
0090 : C7 06 FB 00 FF FF B8 00 B0 E8 1B 00 C7 06 FB 00 : F9
00A0 : 00 00 F7 06 F9 00 04 00 74 06 C7 06 FB 00 FF FF : 3A
00B0 : B8 00 B8 E8 01 00 C3 8E C0 A1 F5 00 D1 E8 D1 E8 : 72
00C0 : D1 E8 8B F0 A1 F7 00 8B D8 D1 E0 D1 E0 03 C3 D1 : 28
00D0 : E0 D1 E0 D1 E0 D1 E0 03 F0 B0 80 8B 0E F5 00 80 : 24
00E0 : E1 07 D2 E8 83 3E FB 00 FF 75 04 26 08 04 C3 F6 : C1
00F0 : D0 26 20 04 C3 00 00 00 00 00 00 00 00 00 00 00 : DD
-----------------------------------------------------------
SUM  : FF B6 79 FD 07 CA 02 49 36 AB 28 71 F8 54 6C E8 : 61
```

CTRL–B

で BASIC に戻り，

DEF　SEG＝&H1F00：A＝0：CALLA ⏎

で実行します。

コマンドメニュー5 (POP〜SAL, SHL)

POP (POP word stack into destination)
ポップ
コード：オペコード表参照
フラグ：変化しない
クロック数：オペコード表参照

POP 命令はスタック上にある**ワードデータ**をレジスタにロードします。たとえば

POP　AX

は細かく考えると，

AX←[SP]
SP=SP+2

ということになります。POP 命令は PUSH 命令とともに使われることが多い命令です。

いま，AX に壊したくないデータが入っているとします。そのとき，サブルーチンを呼ぶ場合

PUSH　AX
CALL　SUBR
POP　AX

とするのが普通です。こうすれば，SUBR で AX レジスタがどのように使われようと，

POP　AX

とした瞬間に AX レジスタの元の値は返ってきます。

POP 命令はスタック上のデータを無条件にもってきますから，スタック上に何が積まれているかを十分理解しておく必要があります。スタックにはサブルーチンのもどりアドレスのオフセット，セグメント値が積まれている場合もありますから，誤ってポップすると暴走する場合もあります。

また，POP 命令はオペコード表のオペランドの項を見れば分かるように，

```
POP　X
```

のように変数にスタック上のデータを直接ロードすることもできます。この場合，変数はワードでなければなりません。また，

```
POP　DS
```

のようにセグメントレジスタにロードすることもできます。これは，次のようにコードセグメントとデータセグメントとを合わせるのに使うことができます。

```
PUSH　CS
POP　 DS
```

POP 命令が**ワード**のオペランドしかとれないことも，よく忘れがちなことです。

```
POP　AL
POP　BYTE PTR [0000H]
```

などは間違いです。

また，8086では複数のレジスタを一度にポップすることはできません。インタラプトの処理ルーチンではレジスタをすべて保存しなければならないことが多いものですが，8086ではこれを行うのに

```
PUSH AX !  PUSH BX
PUSH CX !  PUSH DX
PUSH SI !  PUSH DI
PUSH BP !  PUSH DS
PUSH ES !
          ：インタラプト処理
POP ES !  POP DS
POP BP !  POP DI !  POP SI
POP DX !  POP CX
POP BX !  POP AX
```

などと多くの PUSH，POP 命令を列記するしかないのです。これは8086の欠点といえるかもしれません。この点は8086の上位コンパチブルな80186で改善されています（プッシュオール，ポップオールといった命令が付加されている）。ほかの16ビット CPU，たとえば68000などでは任意のレジスタを同時にプッシ

ュ，ポップできます。

POPF (POP Flags off stack)

フラグのポップ

コード：10011101＝9DH

フラグ：すべて変化

クロック数：8

POPF 命令はスタック上のワードデータをフラグレジスタに読み込みます。つまり

flags ← [SP]

SP＝SP＋2

となります。フラグレジスタと読み込まれるデータとのビットの対応は，

ビット

0　CF（キャリーフラグ）

2　PF（パリティフラグ）

4　AF（補助フラグ）

6　ZF（ゼロフラグ）

7　SF（サインフラグ）

8　TF（トラップフラグ）

9　IF（インタラプトフラグ）

10　DF（ディレクションフラグ）

11　OF（オーバーフローフラグ）

となっています。

PUSH (PUSH word onto stack)

プッシュ

コード：オペコード表参照

フラグ：変化しない

クロック数：オペコード表参照

PUSH 命令はワードデータをスタック上に積みます。PUSH 命令は POP 命令と相補う関係にあるといえます。

たとえば,

    PUSH   AX

では

    SP=SP−2
    [SP] ← AX

となります。先に SP がデクリメントされていることに注意してください。8086では SP は最後に積まれたデータを指しています。

PUSH 命令は POP 命令同様,

    PUSH   X

など，変数をオペランドに指定できます。ただし，ワードデータしかプッシュできません。

    PUSH   AL

は間違いです。

PUSHF (PUSH Flags onto stack)

フラグのプッシュ

コード：10011100＝9CH

フラグ：変化しない

クロック数：10

PUSHF 命令はフラグレジスタの内容をスタックに積みます。各フラグと積まれたデータの各ビットの対応は POPF 命令の項で説明したとおりです。

RCL (Rotate through Carry Left)

キャリーを含めた左ローテイト

コード：オペコード表参照

フラグ：CF，OF が変化

クロック数：オペコード表参照

RCL 命令はオペランドのデータをキャリーとともにローテイトします。具体的な例で考えてみましょう。

AL レジスタ＝00110011B

CF ＝1

だったとします。このとき，

RCL　AL, 1

とすると

AL ＝01110111B

CF ＝0

となります。

つまり，

```
        CF   AL
実行前  1 0 0 1 1 0 0 1 1
実行後  0 0 1 1 0 0 1 1 1
```

となっているわけです。要するに，データが左向きに1ビットだけ回転したような結果となります。

RCL 命令では

RCL　AX, 1

のように1回ローテイトを行うように指定する場合と

RCL　AX, CL

のように **CLレジスタ**の内容の回数だけローテイトを行うように指定する場合とがあります。CL レジスタでローテイトの回数を指定できることは，ほかのローテイトやシフト命令すべてにいえることです。

CL レジスタの内容は変化しません。

MOV　CL, 3

RCL　AX, CL

で AX レジスタは3回ローテイトされますが，CL レジスタの3のままです。

また，CL で回数を指定する場合，時間がかかります。

クロック数

RCL　AX, 1　……2

RCL　AX, CL　……8+4＊N

ですから，速度が要求されるのならば，

RCL　AX, 1!　RCL　AX, 1!　RCL　AX, 1…

のように繰り返し書いたほうがよいでしょう。

また，8086はメモリを直接ローテイトすることができます。つまり，

```
RCL   WORD  PTA (OFFSETARRAY)[SI+BX], CL
RCL   WORD__DATA, 1
```

などと記述できます。

RCR (Rotate through Carry Right)

キャリーを含む右ローテイト

コード：オペコード表参照

フラグ：CF，OF が変化

クロック数：オペコード表参照

RCR 命令はオペランドを右にキャリーとともにローテイトします。回転の方向が左から右になっただけで，ほかは RCR 命令と同じです。

REP/REPE/PEPNE/REPNZ/REPZ (REPeat)

リピート：1111001Z

フラグ：後続のストリング命令を参照

クロック数：6 クロック/ループ

REP 命令は，次に続くストリング命令を繰り返すのに使います。繰り返しの回数は CX レジスタで指定します。

よく使われる例ですが，VRAM (青) を消去するには，

```
MOV    AX, 0A800H
MOV    ES, AX
MOV    CX, 4000H
XOR    AX,AX
XOR    DI, DI
CLD
```

REP　STOSW

で簡単に行うことができます。

REP 命令は CX レジスタの回数だけ繰り返しますが，繰り返しの途中でフラグが変化したときにループから脱出できると便利なこともあります。それを可能とするのが

REPE/REPNE/REPNZ/REPZ

の各命令です。表記上別のものとなっていますが

オペコード

REPE　＝REPZ　＝F3H

REPNE ＝REPNZ ＝F2H

という関係があります。これは

JE　＝JZ

JNE＝JNZ

の関係と同じです。REPE，REPZ はそれぞれ，「イコールの間繰り返せ」「ゼロの間繰り返せ」の意味ですが，CPU 側から見ればまったく同じことになります。

REPE ですが，これは

ZF＝0

ならば繰り返しを終了します。REPE が有効なのは CMPS，SCAS の 2 つの命令実行時だけです。ほかのストリング命令では REP 命令とまったく同じ動作となります。REPE 命令を使う例として，データのサーチがあげられます。バイト配列

DIM　X(999)

の中から 0 でないデータの最初の位置を見つけることを考えましょう。BASIC では，

```
100  FOR  DI=0  TO  999
110    IF  X(DI)<>0THEN  200
120  NEXT  DI
130  すべて0だった
200  DIに位置が入っている
```

となるでしょう。アセンブリ言語では，

```
CLD
MOV    AX, SEG  X
MOV    ES, AX
MOV    DI, OFFSET  X
MOV    CX, 1000
MOV    AL, 0
REPE   SCASB
JNE    FIND  (ループの途中のブレークならば ZF は 0 となっている)
すべて  0
FIND: DI ←データのアドレス+1
```

となります。

REPNE 命令は, REPE の逆で

ZF=0

の間ストリング命令を繰り返します。

ストリング命令は8086の特徴の1つといえますが, それもこの REP プレフィックスとともに使った場合に最大の効果が上がります。

REP 命令で重要なことがあります。それはほかのプレフィックスである LOCK, CS :, ES :, SS : と組み合わせて使う場合です。

```
REP  CS : STOSW
```

などがこれに相当します。これは正常に動作しますが, それは**インタラプトがかからない**場合に限ります。インタラプト処理ルーチンから戻るのに, 8086では1つ前のプレフィックスに戻ってしまいます。つまり,

```
REP
        ←ここに復帰する
CS :
STOSW
```

となるわけです。2つ以上のプレフィックスを使うのであれば, その間**インタラプトを禁止**する必要があります。

PC-9801 では,

キーボードからの割り込み

タイマー割り込み

GDCのVSYNCにより割り込み

などがかかる場合があります。インタラプトを禁止すれば正常に動作するのですが，あまりにも長い時間インタラプトを禁止するのは危険です。インタラプトの禁止はできるだけ短時間にするほうがよいでしょう。

RET (RETurn from procedure)

リターン

コード：オペコード表参照

フラグ：変化しない

クロック数：オペコード表参照

RET命令はサブルーチンからの復帰に使います。CALL命令で呼ばれたサブルーチンは，必ずRETで終わっていなければなりません。

8086にはNEAR CALLとFAR CALLがありましたが，それに応じて

RET，RETF

を使い分ける必要があります。NEAR CALLの場合，スタック上には戻りアドレスの**オフセットしか**積まれていません。だから，RET命令は

IP← [SP]

SP=SP+2

という動作をします。しかし，FAR CALLでは戻り番地の**セグメントとオフセット**がともに積まれているため，RETF命令では

IP← [SP]

SP=SP+2

CS← [SP]

SP+2

とCS，IPの両方が戻ります。ですから，FARコールで呼ばれたのにRETとしたり，NEARコールで呼ばれたのにRETFなどとしたりすれば暴走します。

これは記法上の問題ですが，CP/M-86のASM86では

{ セグメント内リターン　RET
{ セグメント間リターン　RETF

となっているのに対し，MS-DOSのMASMでは

｛セグメント内リターン
　　　　　　　RET, RET　NEAR
　セグメント間リターン　RET　FAR

を使います。

RET 命令でもう1つ重要なことがあります。スタックパラメータを簡単にはずすことができるということです。

コンパイラでは，サブルーチンに引数を渡すときスタックを非常によく使います。たとえば，整数変数X,Yの値をサブルーチン ADDITION に渡す場合，

```
PUSH   Y
PUSH   X
CALL   ADDITION
```

のようにします。サブルーチン側では

```
ADDITION :
            MOV   BP, SP
            MOV   AX, 2[BP]
            ADD   AX, 4[BP]
            RET
```

とすればよいように思えます。

サブルーチン ADDITION が呼ばれたときスタックにはデータが次のように積まれています。

SP→	戻り番地	↑小さな番地
	X	
	Y	↓大きな番地

だから，変数Xの値を AX レジスタにロードするには

```
MOV   AX, SS : 2[SP]
```

としたいところですが，SP レジスタ間接はありませんので，

```
MOV     BP, SP
MOV     AX, SS : 2[BP]
```

とします。ただ，BP レジスタ間接ではデフォルトセグメントが SS ですから

```
MOV     BP, SP
MOV     AX, 2[BP]
```

でよいわけです。BPレジスタ間接のデフォルトセグメントがSSなのは，このようにスタック上のデータを使うためだといえます。

同じように，変数Yの値は

```
4[BP]
```

にありますから

```
ADD     AX, 4[BP]
```

で，AXレジスタに変数XとYの和が求められます。そして，復帰すればよいわけですが，もしここでRET命令を使うと，スタック上に変数X，Yが残ってしまいます。すると，メインルーチン側では

```
PUSH    Y
PUSH    X
CALL    ADDITION
ADD     SP, 4
```

というように，スタックポインタを調整しなければなりません。ところが8086では

```
RET     4
```

といった命令が使えるようになっています。

```
RET     N
```

はサブルーチンから返るときにSPの値にNを加える命令です。動作は

```
IP← [SP]
SP←SP+2+N
```

となります。このように，8086の命令はコンパイラを作るのに都合よくなっています。

ROL (ROtate Left)
ローテイトレフト
コード：オペコード表参照
フラグ：CF，OF が変化
クロック数：オペコード表参照

ROL命令はオペランドをローテイトします。先ほどRCL命令が出てきまし

たが，RCL では

　最上位ビット→ CF

　CF →最下位ビット

となったのに対し，ROL は

　最上位ビット→ CF

　最下位ビット→ CF

とローテイトします。

　AL ＝00110011B

　CF ＝1

のとき

　RCL　AL，1

では

　AL ＝01100111B

　CF ＝0

となりましたが

　ROL　AL，1

では

　AL ＝01100110B

　CF ＝0

となります。これは，左ローテイト実行後，

```
CF        AL
 ┌──────────────┐
 1 0 0 1 1 0 0 1 1 B

 0 0 1 1 0 0 1 1 0 B
```

となるためです。この違いのほかは RCL と同じです。

ROR（ROtate Right）

　ローテイトライト

コード：オペコード表参照

フラグ：CF，OF が変化

クロック数：オペコード表参照

ROR 命令はオペランドを右にローテイトします。ほかは ROL 命令と同じです。

SAHF (Store AH into 8080 Flag)

AH レジスタを8080フラグにストア

コード：10011110＝9EH

フラグ：AF, CF, PF, SF, ZF

クロック数：4

SAHF 命令は AH レジスタの内容を8080フラグ（8086にも8080にもあるフラグ），つまり SF, ZF, AF, PF, CF にストアします。AH レジスタの各ビットとフラグの対応は次のようになります。

BIT 0	CF
BIT 2	PF
BIT 4	AF
BIT 6	ZF
BIT 7	SF

これは8080のプログラムを8086に移植するときに使われることがあります。8080では，

```
PUSH   PSW
POP    PSW
```

という命令がありますが，これを8086に移植するには，単に

```
PUSH PSW   {LAHF
           {PUSH   AX
POP  PSW   {POP    AX
           {SAHF
```

と置き換えればよいのです。

SAL, SHL(Shift Arithmetic Left and SHift logical Left)

シフトアリスメティックレフト

コード：オペコード表参照

フラグ：CF，OF，PF，SF，ZF 変化

　　AF　不定

クロック数：オペコード表参照

SAL，SHL 命令はオペランドを左にシフトします。

　AL＝11001100B

のとき

　SAL　AL，1

を行うと

$$\begin{cases} AL=10011000B \\ CF=1 \end{cases}$$

となります。つまり，

```
       1 1 0 0 1 1 0 0
      ↙ ↙ ↙ ↙ ↙ ↙ ↙ ↙
  CF  1 0 0 1 1 0 0 0←無条件に0
```

CL レジスタでシフト回数が指定できる点はローテイト命令と同じです。

SAL，SHL 命令はシフトにも使えますが，**2のかけ算**にも使えます。「左にシフトすること」は「2をかけること」と同じです。たとえば，

　5 ＊ 2 ＝10

は

　5＝010B→1010B＝10（左シフト）

と同じ結果になります。

8086には乗算命令 MUL，IMUL がありますが，これは非常に時間のかかる命令です（MUL　BXで**118〜133**クロック程度)。それに対し SAL，SHL 命令は

　SAL　AX，1

で**2**クロックと60倍ほど速いのです。2のべき乗の数値をかけるのならば，シフトを使うべきでしょう。また，2のべき乗でなくてもちょっと工夫するとシフトで行えるようになります。たとえば，AX レジスタの内容を5倍するには

　$5 = 2^2 + 1$

ですから

　5 ＊ AX

は

$$5 * AX = 2^2 * AX + AX$$

つまり，

```
MOV   BX, AX
SAL   AX, 1  ⎫
SAL   AX, 1  ⎭ AX=AX * 2
ADD   AX, BX
```

とすればよいのです。定数の乗算は多くの場合，シフトで置き換えたほうがMUL 命令よりもはるかに速くなります。

プログラミングから実行まで

さて，アセンブラ言語によるプログラミング例を示します。「エラトステネスのふるい」で素数を求めるプログラムをとり上げました。

MS-DOS

MS–DOS によるコーディング例だけとします。

リスト2–16のソースをファイル名

OHPC10.ASM

でセーブし，

MASM　OHPC10 ⏎

LINK　OHPC10 ⏎

で実行可能なファイルを作ります。

リスト2-16

```
;
; ８０８６アセンブリ言語講座デモプログラム
;
; 実行方法
; OHPC10.ASM というファイル名でセーブ
; MASM OHPC10;
; LINK OHPC10;
; OHPC10
;
TRUE      EQU       1
FALSE     EQU       0
```

```
STACK1   SEGMENT STACK                          }
         DW      100 DUP (?)                    } スタックの定義
TOP_OF_STACK     LABEL   WORD                   }
STACK1   ENDS

DATA     SEGMENT WORD
FLAGS    DB      8191 DUP(?) ──────────────────── 配列flagsを定義

M1       DB      'エラトステネスのふるい',0DH,0AH,'$'
M2       DB      '  PRIMES',0DH,0AH,'$'
DATA     ENDS                        └── メッセージの終わりは必ず'$'

CODE     SEGMENT
         ASSUME  CS:CODE,DS:DATA,SS:STACK1
START:
         CLI
         MOV     AX,STACK1                      }
         MOV     SS,AX                          } SS, SPの初期化
         MOV     SP,OFFSET TOP_OF_STACK         }
         STI

         MOV     AX,DATA                        } DSの初期化
         MOV     DS,AX

         MOV     AH,09H                         | 「エラトステネスのふるい」を表示
         MOV     DX,OFFSET M1                   | (MS-DOSのシステムコールで行う)
         INT     21H

         MOV     AL,TRUE                        }
         MOV     CX,8191                        }
         MOV     BX,DS                          } 配列の初期化
         MOV     ES,BX                          }     flags [i]=TRUE
         MOV     DI,OFFSET FLAGS                } としている
         CLD                                    }
         REP     STOSB                          }


         MOV     CX,0
         MOV     SI,0
FOR_I:
         CMP     SI,8190
         JG      NEXT_I

         CMP     BYTE PTR (OFFSET FLAGS)[SI],TRUE     flags [i]=TRUE  ?
         JNE     THEN1

         MOV     AX,SI
         SAL     AX,1
         ADD     AX,3
         CALL    PRINT_AX ──────────────────── AXレジスタの内容を10進出力

         CALL    PRINT_SPACE                    }
         CALL    PRINT_SPACE                    } スペースを3つ表示
         CALL    PRINT_SPACE                    }
         MOV     DI,SI
         ADD     DI,AX
FOR_K:
         CMP     DI,8190
         JG      NEXT_K

         MOV     BYTE PTR (OFFSET FLAGS)[DI],FALSE ──── flags[k]=FALSE

         ADD     DI,AX
         JMP     SHORT FOR_K
```

```
NEXT_K:
        INC     CX
THEN1:
        INC     SI
        JMP     FOR_I
NEXT_I:                                          CR, LFを出力
        CALL    CR_LF
        MOV     AX,CX                            } 素数の個数を表示
        CALL    PRINT_AX

        MOV     AH,09H
        MOV     DX,OFFSET M2                     } 「primes」を表示
        INT     21H

        MOV     AH,4CH                           } MS-DOSへ返る
        INT     21H
                                                 以下,AXレジスタを10進出するルーチン
PRINT_AX        PROC    NEAR
        PUSH    AX                               } AX, CXを保存
        PUSH    CX

        MOV     CX,5
        MOV     BX,10
PRINT1:
        CWD
        DIV     BX                               } 各桁を下位からプッシュ
        PUSH    DX

        LOOP    PRINT1
        MOV     CX,5
PRINT2:
        POP     AX
        ADD     AL,'0'
                                                 } 上位の桁から出力
        MOV     AH,2    }
        MOV     DL,AL   } ASCIIコードを出力
        INT     21H     }

        LOOP    PRINT2
        POP     CX
        POP     AX
        RET
PRINT_AX        ENDP ──────────────────── サブルーチンprime_axの終わり

PRINT_SPACE     PROC    NEAR
        PUSH    AX
        MOV     AH,2
        MOV     DL,' '
        INT     21H                              } スペースを出力
        POP     AX
        RET

PRINT_SPACE     ENDP

CR_LF   PROC    NEAR
        MOV     AH,2
        MOV     DL,0DH
        INT     21H
        MOV     AH,2
        MOV     DL,0AH                           } CR, LFを出力
        INT     21H
        RET
CR_LF   ENDP
CODE    ENDS

        END     START
```

コマンドメニュー6（SAR～XOR）

SAR(Shift Arithmetic Right)

右算術シフト

コード：オペコード表参照

フラグ：CF，OF，PF，SF，ZF変化

AF不定

クロック数：オペコード表参照

SAR命令はオペランドを右にシフトします。右シフトにはSHR命令もありますが，SAR命令は符号つきのシフトといえるでしょう。シフト命令というのは，

01010101

というデータがあった場合に

00101010　　　　　　（右シフト）

とすることです。

つまり，

01010101
↘↘↘↘↘↘↘↘
00101010　キャリーへ

のように右に1ビットずらすことになります。SAR命令はオペランドが正か0の場合にいまのような動作をしますが，負の場合（つまり最上位ビットが立っているとき）は，シフト後も最上位ビットは立ったままです。

つまり，

11110000

というデータは

011110000

とはならず，

111110000

となるわけです。一見，妙な動作ですが，これは符号つき除算を行うのにきわ

めて都合がよいのです。

142ページでも述べましたが，シフト命令は2の乗除算に使えます。8086の乗除算命令はきわめて遅いので，極力シフト命令に置き換えるべきですが，SAR命令はこれに使えるのです。

　　−10÷2

の計算を考えてみましょう。

　　−10＝11110110B

ですから，−10にSAR命令を使うと

　　11110110 B

　　11111011 B

となります。

　　11111011B＝−5

ですから，確かに

　　−10÷2＝−5

の計算ができるわけです。除算命令IDIVは，

たとえば

　　IDIV　BL

ならば101〜112クロックもかかりますが，

　　SAR　AL，1

ならば2クロックしかかかりません。SAR命令のほうが約50倍も速いのです。このように，2のべき乗による除算はSAR命令に置き換えたほうがよいでしょう。

また，SAR命令はシフト回数をCLレジスタで指定することもできます。ALレジスタを3回右にシフトするには

　　MOV　CL，3

　　SAR　AL，CL

と書くことができます。ただ，CLレジスタによるシフト回数の指定はけっこう時間がかかります。たとえば，

　　SAR　AL，CL

では，(8＋4×N) クロックかかります (Nはシフト回数)。

　　SAR　AL，1――――――――①

は2クロックですから，速さが問題になる場合は①をシフト回数分書いたほうがよいでしょう。CP/M-86 の ASM86 では何度も書かなければなりませんが，MS-DOS の MASM には REPT マクロという機能があって

```
SAR   AL, 1
SAR   AL, 1
SAR   AL, 1
```

と書くところを

```
REPT   3
SAR   AL, 1
ENDM
```

と書けるので便利です。

SAR 命令のオペランドはオペコード表を見れば分かるように，メモリを直接指定することができます。つまり

```
SAR   X, 1
SAR   BYTE PTR (OFFSET ARRAY)[SI+BX], CL
SAR   WORD PTR[BP], 1
```

などはすべて正しいのです。何度もいうようですが，メモリがレジスタと同じように使えることは便利です。

SBB (SuBtract with Borrow)

ボローを含めた引き算

コード：オペコード表参照

フラグ：AF，CF，OF，PF，SF，ZF 変化

クロック数：オペコード表参照

SBB 命令は引き算を行います。ただ，キャリーフラグが立っている場合，1を余分に引きます。たとえば

```
MOV   AL, 3
SBB   AL, 2
```

の結果は，

CF=0　ならば　AL=3−2=1

ですが，

　　CF＝1　ならば　AL＝3−2−1＝0

となります。

　これも妙な命令ですが，多倍長の引き算に有効な命令です。いま，32ビットどうしの引き算を考えましょう。32ビットの整数データがAX，BXそしてCX，DXに入っているものとします(AX上位，BX下位；CX上位，DX下位)。つまり

AX	BX
CX	DX

（上段から下段を引く：−)）

を行うわけです。

　筆算でするように下の桁から計算します。

```
    SUB   BX, DX
```

で下位16ビットの答がBXに求めることができます。このとき重要なのがキャリーフラグCFの動きです。

　　BX≧DX　ならば　CF＝0

　　BX＜DX　ならば　CF＝1

となります。筆算でも引けない場合は上の桁から借りてきましたが，CPUでも借りたことはCFフラグに残っているのです。この「借り」のことを**ボロー**といいます。さて，上位16ビットは，下位の借りがなければ，

```
    SUB   AX, CX
```

とすればよいのですが，借りがある場合（つまりCFフラグが1の場合）は1を余分に引かなければなりません。つまり，

```
        JB   L1;キャリーが
                      立っていれば L1 へ
        SUB   AX, CX
        JMP   SHORT   L2
    L1: SUB AX, CX
        DEC   AX,      ；余分に1を引く
    L2:
```

と長くなります。しかし，SBB命令があれば，

```
    SBB   AX, CX
```

ですみます。

SCAS(SCAn String)
スキャンストリング
コード：1010111W
フラグ：AF, CF, OF, PF, SF, ZF変化
クロック数：15

SCAS 命令はアキュムレータと ES:[DI] とを比較します。バイト比較，ワード比較があり，それぞれ

SCASB
SCASW

と記述します。SCAS 命令では比較後 DI レジスタを更新します。例によって，

ディレクションフラグ：
DF=0 → インクリメント
DF=1 → デクリメント

となります (SCASB→±1,SCASW→±2)。
SCAS 命令はデータのサーチなどに使います。いま，大きさ LEN のバイト配列 ARRAY_X から`A′をサーチすることを考えます。
プログラムは，

```
CLD
MOV   AX, SEG ARRAY_X
MOV   ES, AX
MOV   DI, OFFSET ARRAY_X
MOV   CX, LEN
MOV   AL, `A′
REPNZ    SCASB
JNZ   NOT_FOUND
DI に`A′の位置が+1が入る
        ⟆
NOT_FOUND:見つからなかった
```

となります。SCASBで

```
    CMP   AL, ES: [DI]
    INC   DI
```

の代わりをしているのです。

このように，SCAS命令は単独で使われるよりもREP，REPNZといったリピートプレフィックスとともに使うことがはるかに多いのです。

SHR(SHift logical Right)

右論理シフト

コード：オペコード表参照

フラグ：CF，OF，PF，SF，ZF変化

AF不定

クロック数：オペコード表参照

SHR命令は右論理のシフトを行います。

SAR命令のように最上位ビットを保存したりしません。たとえば，データは

10110000

はSHRを実行後，

01011000

となります。最上位ビットが保存されないので，SHR命令は2のべき乗の**符号なし除算**に使えます。SHR命令もCLレジスタによりシフト回数を指定できます。

STC(SeT Carry)

キャリーのセット

コード：11111001=F9H

フラグ：CF変化

クロック数：2

STC命令はキャリーフラグCFを1にします。サブルーチンからフラグを返す場合にCFを使うことがありますが，その際STC命令は有効です。

STD(SeT Direction flag)

ディレクションフラグのセット

コード：11111101＝FDH

フラグ：DF 変化

クロック数：2

STD 命令はディレクションフラグ DF をセットします。DF が 1 の場合，ストリング命令実行時にはオートデクリメントモードとなります。

STD

STOSB

では

ES: [DI]←AL

DI←DI－1

となります。DF が 1 だと，インクリメントのような気がしますが，これはデクリメントです。DF の D がデクリメントと覚えると間違いがないと思います（本来は DF は Direction flag の意）。

STI(SeT Interrupt flag)

割り込みフラグをセット

コード：11111011＝ FBH

フラグ：IF 変化

クロック数：2

STI 命令は割り込みフラグを 1 にします。これ以後，8086はインタラプトを受けつけます。STI 命令は CLI 命令とともに使うことがよくあります。あるプログラム，たとえばプログラム A が実行されている間，インタラプトを受けたくない場合，

CLI

⟆

プログラム A

STI

とします。割り込みを禁止する時間はできる限り短くすべきです。

STI 命令で注意することは，STI の次の 1 命令を実行後，割り込みが可能となる点です。これは次のような理由によります。

インタラプト処理ルーチンは

```
CLI
 ⟆
STI
IRET
```

という形になることが多いのですが，インタラプトが頻繁に起こる場合，

```
STI
        ←
IRET
```

と，矢印の部分でインタラプトがかかることもあります。すると，いまのインタラプト処理ルーチンからのもどりアドレスがスタックに積まれたままほかのインタラプト処理が行われます。また，そのルーチンでももどりアドレスが積まれたまま，ほかのインタラプトがかかり…，となる可能性があります。そうすると，スタックを不必要に消費してしまいます。ですから，STI 命令は次の 1 命令の実行を待ってインタラプトの許可を行います。

STOS(STOre String)

ストア　ストリング

コード：1010101W

フラグ：変化しない

クロック数：10

STOS 命令はアキュムレータの内容を ES:［DI］に転送します。転送後，DI レジスタを更新します。バイト転送，ワード転送の違いにより，

```
STOSB
STOSW
```

と記述されます。

STOS 命令はメモリを同一データで埋めるときに有効です。PC-9801 の青の VRAM を消すには

```
CLD
MOV   CX, 4000H
XOR   DI, DI
MOV   AX, 0A800H
MOV   ES, AX
XOR   AX, AX
REP   STOSW
```

ですみます。

SUB(SUBtract)

引き算

コード：オペコード表参照

フラグ：AF, CF, OF, PF, SF, ZF 変化

クロック数：オペコード表参照

SUB 命令は引き算を行います。オペランドとしてメモリイミディエイトが使えるのは便利です。つまり，

```
SUB   X,2
SUB   ARRAY_X[SI],3
```

と，メモリから直接数値が引けるのです。

TEST(TEST)

テスト

コード：オペコード表参照

フラグ：AF 不定

CF, OF＝0

PE, SF, ZF 変化

クロック数：オペコード表参照

TEST 命令はオペランドの論理積をとって，フラグを変化させます。オペランド自体は変化しません。

TEST 命令はオペランドのあるビットが立っているかどうかをテストするのに使えます。たとえば，変数 FLAG のビット 3 が立っているかどうかは，

```
          TEST   FLAG, 8
          JNZ    SET
          立っていない
              ⌇
     SET: 立っている
              ⌇
```

で簡単に調べることができます。また，オペランドが変化しないことは次の場合有効です。

「FLAG のビット 0 が立っていたら L0 に
　　　　ビット 1 が立っていたら L1 に
　　　　ビット 2 が立っていたら L2 に
　飛ぶ」

ような場合で，これは次のようになります。

```
     TEST   FLAG, 1
     JNZ    L1
     TEST   FLAG,2
     JNZ    L2
     TEST   FLAG,4
     JNZ    L3
```

WAIT(WAIT)

ウエイト

コード：10011011＝9BH

フラグ：変化しない

クロック数：最低でも 3

WAIT 命令は，CPUのTEST ピンがアクティブになるまで CPU を「待ち状

態」にします。

WAIT 命令は，外部のハードウェア（大半が8087）と同期するときに使います。

8086と8087は並行的に動作しています。そのために計算速度は上がるのですが，8086が8087の計算結果が必要になった場合，8087の計算終了を待ってデータを得る必要があります。これを実行するのが WAIT 命令なのです。

XCHG(eXCHanGe)

交換

コード：オペコード表参照

フラグ：変化しない

クロック数：オペコード表参照

XCHG 命令はデータの交換に使います。レジスタ AX とレジスタ BX の内容を入れ替えるには

```
XCHG   AX,BX
```

と書きます。変数 X，Y の内容を入れ替えるには，

```
MOV    AX,X
XCHG   AX,Y
MOV    X,AX
```

となります。当然ながら，

```
XCHG   X,Y
```

はできません。

XLAT(transLATe)

トランスレート

コード：11010111＝D7H

フラグ：変化しない

クロック数：11

XLAT 命令は

AL＝[BX＋AL]

という動作をします。つまり，BXレジスタの指すテーブルからAL番目をALに入れます。これはASCIIコードからほかのコードに変換するときに使えます。ALレジスタに入っているASCIIコードの変換は

MOV　BX,OFFSET　TRANS_TABLE XLAT

だけですみます。

XOR(eXclusive OR)

排他的論理和

コード：オペコード表参照

フラグ：CF，OF，PF，SF，ZF変化

AF 不定

クロック数：オペコード表参照

XOR命令はオペランドの排他的論理和をとります。

XORは，レジスタのクリアにもよく使います。AXレジスタをゼロクリアするには

MOV　AX,0―――――――――②

とするのが普通ですが，

XOR　AX,AX―――――――――③

で行うこともできます。③は3クロックしかかからないのでよく使われるのですが，分かりやすさの面で，②のほうがよいでしょう。

プログラミングから実行まで

これで8086のコマンドはすべて解説しました。次はもっと具体的な話を進めていきます。ここでアセンブリ言語のプログラム例を示します。これは，換入法によるソーティングの例です。

BASICではリスト2-17のようになります。

ここではMS-DOSでコーディングしました。リスト2-18をエディタで打ち込み，

リスト2-17

```
1000 N=10
1010 DIM DAT(N)
1020 GOSUB 2000
1030 PRINT "RANDOM DATA :"
1035 GOSUB 3000
1036 PRINT "SROTING START !!"
1040 '
1050 FOR I=2 TO N
1060   WORK=DAT(I)
1070   DAT(0)=WORK
1080   J=I-1
1090   WHILE WORK<DAT(J)
1100     DAT(J+1)=DAT(J)
1110     J=J-1
1120   WEND
1130   DAT(J+1)=WORK
1140 NEXT I
1145 PRINT "SORTING COMPLETED :"
1150 GOSUB 3000
1160 END
2000 '
2010 FOR I=1 TO N
2020  DAT(I)=INT(RND(1)*1000)
2030 NEXT I
2040 RETURN
3000 '
3010 FOR I=1 TO N
3020   PRINT DAT(I);
3030 NEXT I
3034 PRINT
3040 RETURN
```

SORT. ASM

でセーブします。

MASM　SORT ⏎

LINK　SORT ⏎

で実行可能なファイルを作り，

SORT ⏎

で実行します。

リスト2-18

```
;                                            N=100
; SORTING                                    }
;                                            } スタックセグメント
N       EQU     100                          }
STACK1  SEGMENT STACK                        }
        DW      400 DUP (?)
TOP_OF_STACK    LABEL   WORD
STACK1  ENDS
```

```
DATA    SEGMENT WORD                                    ┐
DAT     DW      N DUP (?)    DIM DAT(N)に相当            │
I       DW      ?                                       │
J       DW      ?                                       │
WORK    DW      ?                                       │ データセグメント
SEED    DW      ?                                       │
M1      DB      'RANDOM DATA',0DH,0AH,'$'               │
M2      DB      'SORTING START',0DH,0AH,'$'             │
M3      DB      'SORTING COMPLETED.',0DH,0AH,'$'        │
DATA    ENDS                                            ┘

CODE    SEGMENT BYTE
        ASSUME  CS:CODE,DS:DATA,SS:STACK1
START:
        MOV     AX,DATA                  } DSのセット
        MOV     DS,AX
        CLI
        MOV     AX,STACK1                } SS, SPのセット
        MOV     SS,AX
        MOV     SP,OFFSET TOP_OF_STACK
        STI

        CALL    SET_DAT                  ランダムなデータを配列よりDATにセット
        MOV     AH,9
        MOV     DX,OFFSET M1             「ランダムデータは次のとおり」を表示
        INT     21H

        CALL    PRINT_DAT                配列DATを表示

        MOV     AH,9
        MOV     DX,OFFSET M2             「ソート開始」を表示
        INT     21H

        MOV     I,2                      I=2
SORT1:
        MOV     SI,I
        SHL     SI,1
        MOV     AX,OFFSET DAT[SI]        } WORK=DAT(I)
        MOV     WORK,AX
        MOV     SI,0                     } DAT(0)=WORK
        MOV     OFFSET DAT[SI],AX
        MOV     AX,I
        DEC     AX                       } J=I-1
        MOV     J,AX
SORT2:
        MOV     SI,J
        SHL     SI,1
        MOV     AX,OFFSET DAT[SI]        WORK<DAT(J) ?
        CMP     WORK,AX
        JGE     SORT3
        ADD     SI,2                     } DAT(J+1)=DAT(J)
        MOV     OFFSET DAT[SI],AX
        DEC     J                        J=J-1
        JMP     SHORT SORT2
SORT3:
        MOV     SI,J
        INC     SI
        SHL     SI,1                     DAT(J+1)=WORK
        MOV     AX,WORK
        MOV     OFFSET DAT[SI],AX

        INC     I                        I=I+1
        MOV     AX,I
        CMP     AX,N                     I≦N → SORT1へ
        JLE     SORT1
```

```
        MOV     AH,9                      ┐
        MOV     DX,OFFSET M3              ├「ソート終了…」を表示
        INT     21H                       ┘

        CALL    PRINT_DAT                 配列DATを表示
        MOV     AH,4CH                    ┐
        MOV     AL,0                      ├ MS-DOSへ
        INT     21H                       ┘

PRINT_DAT       PROC    NEAR              ┐
        MOV     SI,OFFSET DAT+2           │
        CLD                               │
        MOV     CX,N                      │
PRINT_D1:                                 │
        LODSW                             │
        PUSH    SI                        │
        PUSH    CX                        ├ 配列DATを表示するルーチン
        CALL    PRINT_AX                  │
        POP     CX                        │
        POP     SI                        │
        LOOP    PRINT_D1                  │
        CALL    PRINT_CR_LF               │
        RET                               │
PRINT_DAT       ENDP                      ┘

SET_DAT PROC    NEAR                      ┐
        MOV     DI,OFFSET DAT+2           │
        CLD                               │
        MOV     AX,DATA                   │
        MOV     ES,AX                     │
        MOV     CX,N                      │
SET_D1:                                   │
        PUSH    DI                        │
        PUSH    CX                        ├ 配列DATにランダムデータを
        CALL    RND1000                   │ セットするルーチン
        POP     CX                        │
        POP     DI                        │
        STOSW                             │
        LOOP    SET_D1                    │
                                          │
        RET                               │
SET_DAT ENDP                              ┘

RND1000 PROC    NEAR                      ┐
        MOV     AX,259                    │
        MUL     SEED                      │
        ADD     AX,3                      │
        AND     AX,32767                  │
        MOV     SEED,AX                   │
        MOV     BX,1000                   ├ 0～999の乱数を返すルーチン
        MUL     BX                        │
        MOV     BX,32767                  │
        DIV     BX                        │
        RET                               │
RND1000 ENDP                              ┘

PRINT_AX        PROC    NEAR              ┐
        MOV     CX,5                      │
        MOV     BX,10                     │
PRINT1:                                   │
        CWD                               │
        DIV     BX                        │
        PUSH    DX                        │
        LOOP    PRINT1                    │
MOV     CX,5
```

```
PRINT2:
        POP     AX
        ADD     AL,'0'

        MOV     AH,2
        MOV     DL,AL
        INT     21H


        LOOP    PRINT2
        CALL    PRINT_SPACE
        CALL    PRINT_SPACE
        CALL    PRINT_SPACE

        RET
PRINT_AX        ENDP

PRINT_CR_LF     PROC    NEAR
        MOV     AH,2
        MOV     DL,0DH
        INT     21H
        MOV     AH,2
        MOV     DL,0AH
        INT     21H
        RET
PRINT_CR_LF     ENDP

PRINT_SPACE     PROC    NEAR
        MOV     AH,2
        MOV     DL,' '
        INT     21H
        RET
PRINT_SPACE     ENDP

CODE    ENDS
        END     START
```

AXレジスタを10進表示するルーチン

CR, LFを出力するルーチン

スペースを出力するルーチン

第3章

ASM86の使い方

ここまでで8086の全命令の解説が終わりました。命令がすべて分かったから，アセンブリ言語でどんどんプログラミングできるかというと，なかなかそうはいかないと思います。理由は

「アセンブリ言語に慣れていない」

「8086のプログラミングに慣れていない」

「アセンブラに慣れていない」

などが考えられます。8ビットCPUでかなりプログラムを書いた人も，8086はなかなか難しいようです。アセンブリ言語が初めての人はなおさらでしょう。何でもそうですが，アセンブリ言語も「慣れること」が上達の早道だと思います。短いプログラムをいくつも書いて何度も暴走させる以外に上達する方法はないでしょう。車と違ってCPUをどんなに暴走させても壊れることはないのですから。

ここでは「各命令が分かる」段階から「なんとかプログラムできる」段階への橋渡し的な解説を行います。

BASICコンパイラになる!?

アセンブリ言語のプログラムを最も簡単に，確実に書く方法は「BASICコンパイラになる」ことでしょう。つまり，作りたいプログラムをまずBASICで組んでみて動作を確認後，それを1対1にアセンブリ言語に直す方法です。

この方法だと，プログラムの論理的な誤りはBASICプログラムのほうで訂正でき，その分アセンブリ言語のデバッグが簡単になります。また，BASICの各命令をアセンブリ言語に書き直すときに間違えなければ，バグなどほとんど出ません。バグが出てもデバッグ時にBASICの各変数の値と機械語の変数の値を照らし合わせれば，どこでミスを犯しているか一目瞭然です。多くの人がこの方法でプログラミングしているはずですが，あまり表立って解説されたことはないので，ここではその解説をしてみましょう。

BASICのあらゆるコマンドをアセンブリ言語に直す方法を解説するのではページ数がいくらあっても足りませんから，必要最小限のコマンドにしたいと思います。

変　数

話を簡単にするため，整数変数と整数配列を扱います。BASICでは変数を使う前に宣言する必要はありませんが，アセンブリ言語ではこれが必要です。整数変数Xを使う場合，CP/M-86のASM86では，

```
X          RW        1
```

とします。MS-DOSのMASMでは

```
X          DW        ?
```

とします。いずれも1ワードの，初期化しないメモリを確保します。これらはデータセグメントに置かなければなりません。

整数配列も同様です。たとえば

```
DIM   A(100)――――――――――①
```

はASM86では

```
A          RW        100+1
```

MASMでは

```
A          DW      100+1   DUP(?)
```

とします。①では

A(0), …, A(100)（101個）

の101ワードのエリアが必要ですから，このように宣言するのです。MASMのDUPは繰り返しを意味し，101個の初期化されないワードメモリを確保します。

代　入

```
X＝1234
```

はアセンブリ言語では

```
MOV       X, 1234
```

とします。

```
X＝Y
```

は

```
MOV       X, Y
```

としたいところですが，8086はメモリからメモリへの転送はできない[注22]ので

```
	MOV	AX, Y
	MOV	X, AX
```

のようにレジスタを介して間接的に代入します。

配列への代入はちょっとやっかいです。

A(10)＝1234

は

```
	MOV	DI, 10 * 2
	MOV	WORD PTR A[DI], 1234
```

となります。

配列の各要素をアクセスするには，このように**レジスタ間接アドレシング**を使うのが自然です。つまり

123[SI], 123[DI], 123[BX]

のように SI,DI,BX, (BP)[注23] レジスタを使って間接的に指定するのです。

A(0)＝1234

は

```
	MOV	DI, 0
	MOV	WORD PTR A[DI], 1234
```

ですが，

A(1)＝1234

はワード配列（1データ2バイト）なので

```
	MOV	DI, 2
	MOV	WORD PTR A[DI], 1234
```

とします。

では，

A(I)＝1234

はどうなるでしょうか？

```
	MOV	DI, I
	MOV	WORD PTR A[DI], 1234
```

ではいけません。ワード配列ですから，DI レジスタには I の2倍が入る必要があります。2倍は乗算命令 MUL を使ってもできますが，2のかけ算ですか

ら**シフト命令**を使うのが普通です。つまり，

```
MOV    DI, I
SHL    DI, I
MOV  A[DI], 1234
```

とします。

加　　算

加算は ADD 命令を使います。

X＝X＋1234

は，

```
ADD    X, 1234
```

とします。8086はメモリに直接加算できることを思い出してください。

X＝X＋Y

は

```
ADD    X, Y
```

とはできません。8086は「メモリにメモリを加えることはできない」のです。これも

```
MOV    AX, Y
ADD    X, AX
```

のように一度レジスタに転送してから加える必要があります。

X＝Y＋Z

は，

```
MOV    AX, Y
ADD    AX, Z
MOV    X, AX
```

となるでしょう。

減　　算

引き算は SUB 命令を使う以外加算と同じです。つまり，

X＝X－1234

　→ SUB　X，1234

X = X − Y

→ { MOV　AX, Y
　 SUB　X, AX

X = Y − Z

→ { MOV　AX, Y
　 SUB　AX, Z
　 MOV　X, AX

となります。

乗　算

かけ算には IMUL 命令を使います。

X = 3 * X

は

```
MOV   AX, 3
IMUL  X
MOV   X, AX
```

となります。IMUL 命令はワード乗算の場合

AX ×[　　]= DX:AX
　　　↑── レジスタ/メモリ

となることに注意してください。かけられる数は AX レジスタに入っている必要があります。また，一般に16ビットデータどうしの乗算結果は**32ビット**になりますから，オーバーフローには注意が必要です。

X = X * Y

は

```
MOV   AX, X
IMUL  Y
MOV   X, AX
```

となります。

除　算

除算には IDIV 命令を使います。

X＝1234÷X

は

```
MOV   AX, 1234
CWD
IDIV  X
MOV   X, AX
```

とします。IDIV 命令はワード除算の場合

```
                    商      余り
DX:AX ÷[     ]＝ AX … DX
          ↑
          └──レジスタ / メモリ
```

という関係になります。割られる数はDX:AXレジスタにセットしなければならないことに注意してください。また，16ビットデータを32ビットデータに直すには **CWD** 命令を使います。

X＝X÷1234

は

```
MOV   AX, X
CWD
MOV   BX, 1234
IDIV  BX
MOV   X, AX
```

とします。8086はイミディエイト値で除算できませんから，このようにBXレジスタなどほかのレジスタにロードして間接的に割ります。

除算で重要なのは0で割らないことです。8086の場合0で割ると内部インタラプトがかかります。このインタラプトの処理ルーチンを自分で書ける人は別ですが，そうでない人は極力0で割らないように注意すべきです。ちなみに，MS-DOSでは0で除算すると

0で除算しました。

というメッセージが出て，プログラムは強制終了させられてしまいます。DISK BASICは何の処理もしません（0による除算のインタラプト処理ルーチンは **IRET** となっています)。

IF～THEN～ELSE

条件分岐はCMP命令と条件ジャンプで実現します。

IF　X＞0　THEN ～

は

```
        CMP   X, 0
        JG    L1
        JMP   L2
L1:
          ⋮
L2:
```

となります。また，

IF　X＞0 THEN～ELSE …

は

```
        CMP   X, 0
        JG    L1
        JMP   L2
L1:
        JMP   L3
L2:
          ⋮
L3
          )
```

となります。

IF　X＞Y　THEN …

の場合は

```
        MOV   AX, X
        CMP   AX, Y
        JG    L1
        JMP   L2
L1:
```

```
        ⋮
  L2:
```

のように AX レジスタなどに転送してから比較します。CMP 命令はメモリどうしの比較ができないことに注意してください。

>，≧，＝，≦，<，≠

と条件ジャンプ命令の対応は表3-1のようになります。

表3-1

＝	JE/JZ	≧	JGE/JNL	≦	JLE/JNG
>	JG/JNLE	<	JL/JNGL	≠	JNE/JNZ

```
  IF   X > 0   THEN …
```

の場合

```
      CMP    X, 0
      JLE    L1
        ⋮
  L1:
```

のように条件を逆にすれば短くなりますが間接ジャンプは＋127～－128しか飛べないため，THEN 以下の処理が125バイト以上になると

label out of range!

のエラーとなります。初めのうちは最初のようにコーディングすることをお勧めします。9801はメモリが十分あるのですから，2バイト程度短くしたところで何の得にもなりません。

FOR～NEXT

FOR～NEXT 文は条件分岐命令で書き換えます。

```
  FOR   I = 1   TO 100   STEP2
        ⋮
  NEXT    I
```

はBASICのIF～THEN～ELSEならば

```
    I = 1
  *L1
    IF I ≦100 THEN  *L2 ELSE  *L3
  *L2
        ⋮
      I = I + 2
      GOTO    *L1
  *L3
```

と書き換えられますから，アセンブリ言語でも

```
      MOV    I, 1
  L1:
      CMP    I, 100
      JLE    L2
      JMP    L3
  L2:
        ⋮
      ADD    I, 2
      JMP    L1
  L3:
```

となります。

```
    FOR I=S TO E STEP ST
        ⋮
    NEXT I
```

の場合は

```
      MOV   AX, S    } I = S
      MOV   I, AX
  L1: MOV   AX, I    } I ≦ E ?
      CMP   AX, E
      JLE   L2
      JMP   L3
```

```
L2:       ⋮
      MOV    AX, ST   } I = I + ST
      ADD    I, AX    }
      JMP    L1
```

です。8086では**メモリとメモリ**の加算，比較，代入はできません。上のように一度レジスタにロードしてから行います。

入出力

入出力，いわゆるINPUT，PRINT命令などはやっかいです。これはハードウェアによりプログラムが違ってきます。ただ，MS-DOSやCP/M-86などでは1文字入出力はシステムコールとして用意されているので，これを利用すればいいでしょう。

BASICのINPUT，PRINT命令はかなり柔軟にデータの入出力ができるようになっているので，これとまったく同じ機能をもたせるのは大変です。これらは私個人としてはOSがもつべき機能だと思いますが，残念ながらMS-DOSもCP/M-86も1文字入出力程度のローレベルな入出力しかもっていません。たとえばCP/M-86では

```
    MOV    CL, 9
    MOV    DX, OFFSET MESSAGE
      INT       224
        ⋮
MESSAGE DB  "This is message. $"
```

で文字例

    This is message.

が出力されます。

では，いままでに述べた方法でプログラムを書いてみましょう。

円周率πを求めるプログラム

πを求める有名な式に

$$\pi = 16\tan^{-1}(1/5) - 4\tan^{-1}(1/239)$$

（マーチンの公式）

があります。$\tan^{-1}x$を展開した総和によってπを求めるわけです。BASICで書くとリスト3-1のようになるでしょう。1010行のMAXの値を変えることにより，100桁でも1000桁でもπを求めることができますが，BASICだとどれほどの時間がかかるか分かりません。これをそのままの形でアセンブリ言語に直すと，リスト3-2のようになります。まったく最適化は行っていませんが，2000桁を求めるのに33分（PC-9801）ほどしかかかりません。

リスト3-1

```
1000 DEFINT A-Z
1010 MAX=10
1020 DIM X(MAX),Y(MAX),ATN.(MAX),A(MAX),B(MAX),C(MAX),D(MAX)
1030 N.ATN=16:D.ATN=5:GOSUB *ARCTANGENT
1040 FOR I=0 TO MAX:C(I)=ATN.(I):NEXT I
1050 N.ATN=4:D.ATN=239:GOSUB *ARCTANGENT
1060 FOR I=0 TO MAX:D(I)=ATN.(I):NEXT I
1070 FOR I=0 TO MAX:X(I)=C(I):Y(I)=D(I):NEXT I
1080 GOSUB *SUBTRACTION
1090 PRINT USING "PI=#.";X(0);
1100 FOR I=1 TO MAX:PRINT USING "#";X(I);:NEXT I
1110 PRINT:END
1115 '
1120 *ADDITION
1130  CARRY=0
1140 FOR I=MAX TO 0 STEP -1
1150    TEMP=X(I)+Y(I)+CARRY
1160    X(I)=TEMP MOD 10                    加算
1170    CARRY=TEMP¥10                         X( )=X( )+Y( )
1180 NEXT I
1190 RETURN
1200 '
1210 *SUBTRACTION
1220 BORROW=1
1230 FOR I=MAX TO 0 STEP -1
1240    TEMP=X(I)-Y(I)+9+BORROW             減算
1250    X(I)=TEMP MOD 10                      X( )=X( )-Y( )
1260    BORROW=TEMP¥10
1270 NEXT I
1280 RETURN
1290 '
1300 *DIVISION
1310 REMAINDER=0
1320 FOR I=0 TO MAX                         除算
1330    TEMP=10*REMAINDER+X(I)                X( )=X( )/N.DIV
1340    X(I)=TEMP¥N.DIV
```

```
1350   REMAINDER=TEMP MOD N.DIV
1360 NEXT I
1370 RETURN
1380 '
1390 *ARCTANGENT
1400 X(0)=N.ATN:FOR I=1 TO MAX:X(I)=0:NEXT I
1410 N.DIV=D.ATN:GOSUB *DIVISION
1420 FOR I=0 TO MAX:A(I)=X(I):ATN.(I)=X(I):NEXT I
1430 J=3:K=0
1440 *ATN.LOOP
1450  IF K>MAX THEN RETURN
1460 IF A(K)=0 THEN K=K+1:GOTO *ATN.LOOP
1470 FOR I=0 TO MAX:X(I)=A(I):NEXT I:N.DIV=D.ATN
1480  GOSUB *DIVISION:GOSUB *DIVISION
1490  FOR I=0 TO MAX:A(I)=X(I):NEXT I
1500  N.DIV=J:GOSUB *DIVISION
1510  FOR I=0 TO MAX:B(I)=X(I):NEXT I
1520  FOR I=0 TO MAX:X(I)=ATN.(I):Y(I)=B(I):NEXT I
1530  IF(J MOD 4)=1 THEN GOSUB *ADDITION ELSE GOSUB *SUBTRACTION
1540  FOR I=0 TO MAX:ATN.(I)=X(I):NEXT I
1550  J=J+2
1560 GOTO *ATN.LOOP
```

(1390～1560) $\tan^{-1}$の計算
ATN.( )
= N.ATN *
$\tan^{-1}$(1/D.ATN)

リスト3-2

```
STACK1  SEGMENT STACK
        DW      200 DUP(?)
TOS     LABEL   WORD
STACK1  ENDS

DATA    SEGMENT WORD
;1000 DEFINT A-Z
;1010 MAX=100
MAX     =       100
;1020 DIM X(MAX),Y(MAX),ATN.(MAX),A(MAX),B(MAX),C(MAX),D(MAX)
X       DW      MAX+1   DUP(?)
Y       DW      MAX+1   DUP(?)
ATN     DW      MAX+1   DUP(?)
A       DW      MAX+1   DUP(?)
B       DW      MAX+1   DUP(?)
C       DW      MAX+1   DUP(?)
D       DW      MAX+1   DUP(?)

N_ATN   DW      ?
D_ATN   DW      ?
N_DIV   DW      ?
I       DW      ?
J       DW      ?
K       DW      ?
CARRY   DW      ?
BORROW  DW      ?
TEMP    DW      ?
REMAINDER       DW      ?

DATA    ENDS

CODE    SEGMENT BYTE
        ASSUME  CS:CODE,DS:DATA,SS:STACK1
START:
        CLI
        MOV     AX,STACK1
```

(STACK1～STACK1 ENDS) スタックセグメントの定義

(DATA SEGMENT) 以下，データセグメント

(X～D) アセンブリ言語による配列．BASICでは
DIM X(100)
ならば
X(0)，X(1)，…，X(100)
101個
なので(MAX+1)個のエリアを確保

(N_ATN～REMAINDER) 各変数の宣言

(CODE SEGMENT) 以下，コードセグメント

(CLI) 始め

```
        MOV     SS,AX                   スタックセグメント，スタックポインタ
        MOV     SP,OFFSET TOS           の初期化
        STI

        MOV     AX,DATA                 データセグメントの設定
        MOV     DS,AX
; 1030 N.ATN=16:D.ATN=5:GOSUB *ARCTANGENT
        MOV     N_ATN,16                配列
        MOV     D_ATN,5                   ATN(0), ATN(1), …, ATN(MAX)
        CALL    ARCTANGENT              に16tan⁻¹(1/5)をセット
;1040 FOR I=0 TO MAX:C(I)=ATN.(I):NEXT I
        MOV     I,0
FOR1:
        CMP     I,MAX
        JG      NEXT1

        MOV     SI,I                    配列ATNを配列Cにコピー
        SHL     SI,1
        MOV     AX,OFFSET ATN[SI]
        MOV     OFFSET C[SI],AX
        INC     I
        JMP     SHORT   FOR1
NEXT1:
;1050 N.ATN=4:D.ATN=239:GOSUB *ARCTANGENT
        MOV     N_ATN,4
        MOV     D_ATN,239               配列ATNに 4 tan⁻¹(1/239)をセット
        CALL    ARCTANGENT
;1060 FOR I=0 TO MAX:D(I)=ATN.(I):NEXT I
        MOV     I,0
FOR2:
        CMP     I,MAX
        JG      NEXT2
        MOV     SI,I                    配列ATNを配列Dにコピー
        SHL     SI,1
        MOV     AX,OFFSET ATN[SI]
        MOV     OFFSET D[SI],AX
        INC     I
        JMP     SHORT   FOR2
NEXT2:
;1070 FOR I=0 TO MAX:X(I)=C(I):Y(I)=D(I):NEXT I
        MOV     I,0
FOR3:
        CMP     I,MAX
        JG      NEXT3
        MOV     SI,I
        SHL     SI,1
        MOV     AX,OFFSET C[SI]         配列Cを配列Xに，配列Dを配列Yにコピー
        MOV     OFFSET X[SI],AX

        MOV     AX,OFFSET D[SI]
        MOV     OFFSET Y[SI],AX

        INC     I
        JMP     SHORT   FOR3
NEXT3:
;1080 GOSUB *SUBTRACTION                X＝X－Y(Xにπがセットされる)
        CALL    SUBTRACTION
;1090 PRINT USING "PI=#.";X(0);
        MOV     AL,'P'
        CALL    ONCOUT
        MOV     AL,'I'
        CALL    ONCOUT
        MOV     AL,'='                  'PI='とX(0)と' 'を表示
        CALL    ONCOUT
        MOV     A> ·························X(0)=X
        CALL    PRINT_AL
```

```
        MOV     AL,'.'
        CALL    ONCOUT
;1100 FOR I=1 TO MAX:PRINT USING "#";X(I);:NEXT I
        MOV     I,1
FOR4:
        CMP     I,MAX
        JG      NEXT4

        MOV     SI,I                          配列の残り
        SHL     SI,1                            X(1), X(2), …X(MAX)
        MOV     AX,OFFSET X[SI]               を表示
        CALL    PRINT_AL

        INC     I
        JMP     SHORT FOR4
NEXT4:
;1110 PRINT:END
        MOV     AL,0DH
        CALL    ONCOUT                        CR, LFを出力
        MOV     AL,0AH
        CALL    ONCOUT

        MOV     AH,4CH
        MOV     AL,0                          MS-DOSへ
        INT     21H

;1115                                         加算ルーチン
; 1120 *ADDITION                              「X=X+Y」
ADDITION        PROC    NEAR
;1130 CARRY=0
        MOV     CARRY,0
;1140 FOR I=MAX TO 0 STEP -1
        MOV     I,MAX
FOR5:
        CMP     I,0
        JL      NEXT5
;1150   TEMP=X(I)+Y(I)+CARRY
        MOV     SI,I
        SHL     SI,1
        MOV     AX,OFFSET X[SI]
        ADD     AX,OFFSET Y[SI]
        ADD     AX,CARRY
        MOV     TEMP,AX
;1160   X(I)=TEMP MOD 10
;1170   CARRY=TEMP¥10
        MOV     AX,TEMP
        CWD
        MOV     BX,10
        IDIV    BX                            IDIV BX
        MOV     OFFSET X[SI],DX               の結果がAXに,余りがDXに入っている
        MOV     CARRY,AX
;1180 NEXT I
        DEC     I
        JMP     SHORT FOR5
NEXT5:
;1190 RETURN
        RET
ADDITION        ENDP

;1200
;1210 *SUBTRACTION                            減算ルーチン
SUBTRACTION     PROC    NEAR                  「X=X-Y」
;1220 BORROW=1
        MOV     BORROW,1
;1230  FOR I=MAX TO 0 STEP -1
        MOV     I,MAX
```

```
FOR6:
        CMP     I,0
        JL      NEXT6
;1240   TEMP=X(I)-Y(I)+9+BORROW
        MOV     SI,I
        SHL     SI,1

        MOV     AX,OFFSET X[SI]
        SUB     AX,OFFSET Y[SI]
        ADD     AX,9
        ADD     AX,BORROW
        MOV     TEMP,AX
;1250   X(I)=TEMP MOD 10
;1260   BORROW=TEMP¥10
        MOV     AX,TEMP
        CWD
        MOV     BX,10
        IDIV    BX
        MOV     OFFSET X[SI],DX
        MOV     BORROW,AX
;1270 NEXT I
        DEC     I
        JMP     SHORT   FOR6
NEXT6:
;1280 RETURN
        RET
SUBTRACTION     ENDP

;1290
;1300 *DIVISION
DIVISION        PROC    NEAR
;1310 REMAINDER=0
        MOV     REMAINDER,0
; 1320   FOR I=0 TO MAX
        MOV     I,0
FOR7:
        CMP     I,MAX
        JG      NEXT7
;1330    TEMP=10*REMAINDER+X(I)
        MOV     AX,10
        IMUL    REMAINDER
        MOV     SI,I
        SHL     SI,1
        ADD     AX,OFFSET X[SI]
        MOV     TEMP,AX
;1340   X(I)=TEMP¥N.DIV
;1350   REMAINDER=TEMP MOD N.DIV
        MOV     AX,TEMP
        CWD
        IDIV    N_DIV
        MOV     OFFSET X[SI],AX
        MOV     REMAINDER,DX
;1360 NEXT I
        INC     I
        JMP     SHORT   FOR7
NEXT7:
;1370   RETURN
        RET
DIVISION        ENDP
```

除算ルーチン
「X = X/N_DIV」

```
;1380 '
;1390 *ARCTANGENT
ARCTANGENT      PROC    NEAR
;1400 X(0)=N.ATN:FOR I=1 TO MAX:X(I)=0:NEXT I
        MOV     AX,N_ATN
        MOV     X,AX
```

$\tan^{-1}$計算ルーチン
「ATN＝N_ATN・$\tan^{-1}$(1/N_DIV)」

```
        MOV     I,1
FOR8:
        CMP     I,MAX
        JG      NEXT8
        MOV     SI,I
        SHL     SI,1
        MOV     WORD PTR (OFFSET X)[SI],0

        INC     I
        JMP     SHORT FOR8
NEXT8:
;1410 N.DIV=D.ATN:GOSUB *DIVISION
        MOV     AX,D_ATN
        MOV     N_DIV,AX
        CALL    DIVISION
;1420 FOR I=0 TO MAX:A(I)=X(I):ATN.(I)=X(I):NEXT I
        MOV     I,0
FOR9:
        CMP     I,MAX
        JG      NEXT9

        MOV     SI,I
        SHL     SI,1

        MOV     AX,OFFSET X[SI]
        MOV     OFFSET A[SI],AX
        MOV     OFFSET ATN[SI],AX

        INC     I
        JMP     SHORT   FOR9
NEXT9:
;1430 J=3:K=0
        MOV     J,3
        MOV     K,0
;1440 *ATN.LOOP
ATN_LOOP:
;1450    IF K>MAX THEN RETURN
        CMP     K,MAX
        JLE     ATN1
        RET
ATN1:
;1460    IF A(K)=0 THEN K=K+1:GOTO *ATN.LOOP
        MOV     SI,K
        SHL     SI,1
        CMP     WORD PTR (OFFSET A)[SI],0
        JNE     ATN2
        INC     K
        JMP     SHORT   ATN_LOOP
ATN2:
;1470    FOR I=0 TO MAX:X(I)=A(I):NEXT I:N.DIV=D.ATN
        MOV     I,0
FOR10:
        CMP     I,MAX
        JG      NEXT10

        MOV     SI,I
        SHL     SI,1

        MOV     AX,OFFSET A[SI]
        MOV     OFFSET X[SI],AX

        INC     I
        JMP     SHORT FOR10
NEXT10:
        MOV     AX,D_ATN
        MOV     N_DIV,AX
```

X(0)=N_ATN
X(1), …, X(MAX)=0

X=X/D_ATN

配列Xを配列A, ATNにコピー

配列Aを配列Xにコピー

```
; 1480 GOSUB *DIVISION:GOSUB *DIVISION                  X＝X/D_ATN/D_ATN
        CALL     DIVISION
        CALL     DIVISION
;1490   FOR I=0 TO MAX:A(I)=X(I):NEXT I
        MOV      I,0
FOR11:
        CMP      I,MAX
        JG       NEXT11

        MOV      SI,I
        SHL      SI,1                              配列Xを配列Aにコピー

        MOV      AX,OFFSET X[SI]
        MOV      OFFSET A[SI],AX

        INC      I
        JMP      SHORT   FOR11
NEXT11:
;1500   N.DIV=J:GOSUB *DIVISION
        MOV      AX,J
        MOV      N_DIV,AX                          X＝X/J
        CALL     DIVISION
;1510   FOR I=0 TO MAX:B(I)=X(I):NEXT I
        MOV      I,0
FOR12:
        CMP      I,MAX
        JG       NEXT12

        MOV      SI,I
        SHL      SI,1                              配列Xを配列Bにコピー

        MOV      AX,OFFSET X[SI]
        MOV      OFFSET B[SI],AX

        INC      I
        JMP      SHORT FOR12
NEXT12:
;1520   FOR I=0 TO MAX:X(I)=ATN.(I):Y(I)=B(I):NEXT I
        MOV      I,0
FOR13:
        CMP      I,MAX
        JG       NEXT13

        MOV      SI,I
        SHL      SI,1

        MOV      AX,OFFSET ATN[SI]                 配列ATNを配列Xに,
        MOV      OFFSET X[SI],AX                   配列Bを配列Yにコピー

        MOV      AX,OFFSET B[SI]
        MOV      OFFSET Y[SI],AX

        INC      I
        JMP      SHORT FOR13
NEXT13:
;1530   IF(J MOD 4)=1 THEN GOSUB *ADDITION ELSE GOSUB *SUBTRACTION
        MOV      AX,J
        AND      AX,3
        CMP      AX,1
        JNE      ATN3

        CALL     ADDITION                          X＝X＋Y
        JMP      SHORT ATN4
ATN3:
        CALL     SUBTRACTION                       X＝X－Y
ATN4:
```

```
;1540    FOR I=0 TO MAX:ATN.(I)=X(I):NEXT I
         MOV     I,0
FOR14:
         CMP     I,MAX
         JG      NEXT14

         MOV     SI,I
         SHL     SI,1

         MOV     AX,OFFSET X[SI]
         MOV     OFFSET ATN[SI],AX

         INC     I
         JMP     SHORT   FOR14
NEXT14:
;1550    J=J+2
         ADD     J,2
;1560    GOTO *ATN.LOOP
         JMP     ATN_LOOP
ARCTANGENT       ENDP

ONCOUT   PROC    NEAR
         MOV     AH,02
         MOV     DL,AL
         INT     21H
         RET
ONCOUT   ENDP

PRINT_AL         PROC    NEAR
         ADD     AL,'0'
         CALL    ONCOUT
         RET
PRINT_AL         ENDP

CODE     ENDS

         END     START
```

配列Xを配列ATNにコピー

ALをASCII表示するルーチン

ALを10進出力するルーチン

ここではMS-DOSによってコーディングします。ソースリストをPI.ASMでセーブし,

MASM PI; ⊘

LINK PI; ⊘

で実行可能なファイルを作ります。

実行結果を図3-1に示します。最後の1桁は誤差を含んでいます。

BASICで書いたプログラムをアセンブリ言語に書き換える方法ではバグはほとんど出ません。事実PI.ASMを書いたときもバグは出ませんでした。

BASICをアセンブリ言語に直す方法では,冗長なプログラムになることは避けられませんが,最初のうちはそんなことは気にせず,動くプログラムを書いたほうがよいと思います。そのうち「この部分はこのように書いたほうが速い」といったことも分かってくるでしょう。とにかく小さな動くプログラムを

図3-1

```
現在の時刻は 10:11:34.00 です
PI=3.141592653589793238462643383279502884197169399375105820974944592307816406286
20899862803482534211706798214808651328230664709384460955058223172535940812848111
74502841027019385211055596446229489549303819644288109756659334461284756482337867
83165271201909145648566923460348610454326648213393607260249141273724587006606315
58817488152092096282925409171536436789259036001133053054882046652138414695194151
16094330572703657595919530921861173819326117931051185480744623799627495673518857
52724891227938183011949129833673362440656643086021394946395224737190702179860943
70277053921717629317675238467481846766940513200056812714526356082778577134275778
96091736371787214684409012249534301465495853710507922796892589235420199561121290
21960864034418159813629774771309960518707211349999998372978049951059731732816096
31859502445945534690830264252230825334468503526193118817101000313783875288658753
32083814206171776691473035982534904287554687311595628638823537875937519577818577
80532171226806613001927876611195909216420198938095257201065485863278865936153381
82796823030195203530185296899577362259941389124972177528347913151557485724245415
06959508295331168617278558890750983817546374649393192550604009277016711390098488
24012858361603563707660104710181942955596198946767837449448255379774726847104047
53464620804668425906949129331367702898915210475216205696602405803815019351125338
24300355876402474964732639141992726042699227967823547816360093417216412199245863
15030286182974555706749838505494588586926995690927210797509302955321165344987202
75596023648066549911988183479775356636980742654252786255181841757467289097777279
38000816470600161452491921732172147723501414419735685481613611573525521334757418
49468438523323907394143334547762416862518983569485562099219222184272550254256887
67179049460165346680498862723279178608578438382796797668145410095388378636095068
00642251252051173929848960841284886269456042419652850222106611863067442786220391
94945047123713786960956364371917287467764657573962413890865832645995813390478027
59003
現在の時刻は 10:45:38.00 です
```

いくつも作ってみてください。

8086の命令の説明，簡単なプログラムの書き方（BASICコンパイラによる）が終わったので，これからは実際のアセンブラを使っていくことを考えます。8086の全命令が分かって一応のプログラムが組めるようになっても，実際に動かしてみるとなかなか思うように動いてくれないものです。理由としては，

- アセンブラの文法に慣れていない
- 使っているOSに慣れていない

などが考えられます。

一般に使われている8086用アセンブラとしては，

CP/M-86…ASM86, RASM86[注24]

MS-DOS…MASM

があげられます。それぞれ一長一短がありますが，機能の面から見れば

MASM＞RASM86＞ASM86

アセンブル速度から見ると

ASM86＞RASM86＞MASM

といえるでしょう。

これからはASM86をとり上げます。CP/M-86のASM86は

●アセンブルが高速である

●シンボリックデバッグできる

などMS-DOSのMASMよりよい点もあり，比較的短いプログラムならば十分実用になるアセンブラです。特に，SID86でシンボリックデバッグできることは重要です。シンボリックデバッガとは，デバッグ時にアドレスや変数を数値で示すのではなく，ラベル名や変数名で示してくれます。

たとえば，

```
        ORG  100H
  TEST:XOR  CL, CL
        XOR  DL, DL
        INT  224
        END
```

といったプログラムを実行するときに，DDT86では

G100　　　(100H番地から実行せよ)

と数値で100とアドレスを示すだけでしたが，シンボリックデバッグSID86ならば，これに加えて，

G. TEST (ラベル TEST から実行せよ)

というように，名前で指定することができます。

このように，シンボリックデバッガはきわめて強力なツールで，これがあるのとないのとでは開発効率は数段違うといっても過言ではないでしょう。ASM86で書いたプログラムはSID86でデバッグできるわけで，その点だけでもASM86は使う価値があるといえます。MS-DOSにはSID86に相当するプログラムはありません。ですからMASMで作ったプログラムは

DEBUG

という平凡なデバッガでデバッグするしかありません。

これからCP/M-86のアセンブラASM86の解説をするわけですが，ASM86のマニュアルは直訳のためか非常に分かりにくく，アセンブラが初めての人には不親切です。ここでは厳密な説明よりも，とにかく使えることを主眼において解説してみます。

最も短いASM86のプログラム

CP/M-86上で動作する最も簡単なプログラムは，次のようになります。

```
CSEG
ORG   100H
RETF
END
```

これをファイルTEST. A86にセーブしたとすると，

```
ASM86 TEST  ⏎
GENCMD TEST 8080 ⏎
```

（8080には注25）

で実行可能な

```
TEST.CMD
```

ができ上がります。

TEST. A86は単にCP/M-86に返るだけのプログラムですが，重要な要素を含んでいます。まず，

「コードは CSEG の後ろに置かなければならない」

ということです。8086ではセグメントが重要ですが，以下のデータがコードセグメントに属することを示すのが**CSEG**です。

CSEG は8086の命令ではなく ASM86 への命令です。このように，CPU の命令と同じようにプログラム中に書かれアセンブラに命令するものを**擬似命令**と呼びます。アセンブラでプログラムを組むには

- CPU の命令
- アセンブラの擬似命令

の両方を知る必要があります。ただ，擬似命令の数はそれほど多くなく，すべてを知る必要もないので恐れることはありません。また，最低限必要な擬似命令はここで解説してしまいます。

TEST. A86から分かることの第2は，

「コードの先頭は100H番地から始めなければならない」

ことです。これを指定しているのが

```
ORG    100H
```

です。ORG[注26]もやはり擬似命令でオフセットアドレスを指定するのに使います。たとえば,

ORG　200H

TEST:

とすれば,ラベルTESTのオフセット値は200Hとなります。どうして100H番地から始めるのでしょう。その理由は,プログラムをCP/M-86上で動作させるためです。CP/M-86では0〜FFH番地を**ベースページ**と呼び,DMA[注27]やFCB[注28]などに使っています。仮に

ORG　0000H

でプログラムを書き始めると,オブジェクトは得られますが,CP/M-86上で実行する際に0〜FFHが破壊されて,プログラムはうまく動きません。

次にTEST.A86から分かることは,

「CP/M-86に返るには RETF を使う」

ことです。CP/M-86のCCP[注29]はプログラムを適当なメモリ位置にロードしたあと,**ファーコール**でプログラムに飛び込んできます。ですからCCPに返るには,**ファーリターン**であるRETFを使うのです。プログラムからの抜け方が分からない人がけっこういるので覚えておいてください。

最後にTEST.A86から分かるのは

「プログラムの終わりは END である」

ことです。ENDも擬似命令で,プログラムの終わりを表します。ASM86ではENDはなくてもかまいません。そのときは,ファイルの終わり(EOF=1AH)をプログラムの終わりとみなします。

普通のASM86のプログラム

前項ではコードしかない例を示しました。普通のプログラムにはデータもあります。ASM86で，データも含む一般的なプログラムは次のようになります。

```
            CSEG ─────────────── ①
            ORG    100H ──────── ②
            [プログラム]
            RETF ─────────────── ③
    END_CS:
            DSEG
            ORG OFFSET  END_CS
               [データ]
            END
```

①～③についてはもう説明しました。

END_CS

はラベルです。ASM86のラベルはどんなに長くてもかまいません。[注30] 8ビットCPU用アセンブラのラベル名は6文字程度で，うまいラベル名が思い浮かばないことも多いものですが，ASM86ではそんなことはありません。ラベル名に使えるのはA～Z，a～z，0～9と_（アンダーバー），@（アットマーク）です。特に，アンダーバーは長いラベル名を読みやすくするのに多用されます。

たとえば，

```
SEND_MIDI_DATA:
OPEN_FILE:
PRINT_STRING:
```

などです。なお，_（アンダーバー）はアセンブラに無視されるので注意が必要です。つまり

```
PRINT_STRING:
PRINTSTRING:
```

の2つは同じラベルとみなされます。

6文字程度のラベル名しか受けつけないアセンブラのユーザーはASM86などのソースプログラムを見て

「ラベル名が長すぎる！」

とガタガタいいますが，ラベル名が長いことは決して悪いことではありません。

ONCOUT

と

ONE_CHARACTER_OUTPUT

とでは後者のほうがはるかに分かりやすいはずです。

次に，

DSEG

ですが，これは以下がデータセグメントに属していることを示します。

次の

ORG OFFSET END_CS

はコードの直後からデータを配置することを指定します。ORGの後ろには数値がきます（例：ORG　100H，ORG　200H)。いま，コードの最後はラベルEND_CS: になっていますから

ORG　END_CS

としたいところですが，END_CSはラベルのためORGのオペランドとはなりえません。必要なのはラベルではなくラベルのオフセット値です。ラベルからオフセット値を求める演算子に

OFFSET

があります。たとえば

```
        ORG    1234H
TEST:
```

というプログラムで，

OFFSET TEST

は数値

1234H

を表します。だから，

ORG OFFSET END_CS

でコードの直後からデータを始めることができます。最初のうちは

```
END_CS:
        DSEG
        ORG OFFSET END_CS
```

と書くのだと覚えたほうがよいでしょう。この後ろに変数や配列を書けばよいのです。つまり,

```
        DSEG
        ORG OFFSET END_CS
X       DB  12
Y       DW  1234
ARRAY   RB  100
LONG_VARIABLE_NAME DB 123
        END
```

などとなるでしょう。

```
X    DB    12
```

はバイト変数Xを宣言して,その初期値を12とします。

DB(Define Byte) は擬似命令で,バイト変数を定義するのに使います。

```
Y    DW    1234
```

はワード変数Yを宣言して,初期値を1234とします。DW(Define Word) も擬似命令でワード変数を定義するのに使います。

```
ARRAY    RB    100
```

は100バイトの大きさのバイト配列 ARRAY を定義します。初期化はされません。RB も擬似命令でオペランドで示されたバイト数のメモリを確保し,バイト変数を定義します。RB はいまの例のように,バイト配列を宣言するのに使います。ワード配列なら,

```
ARRAY    RW    100
```

のように RW 擬似命令を使います。

```
LONG_VARIABLE_NAME    DB 123
```

は長い変数名の例です。ASM86では変数名にも長さの制限がありません。また使える文字もラベルも同様です。

実際の短いプログラム 5

ここまでが分かっていれば，ASM86のプログラムは一応作れるはずです。ここで簡単なプログラムを作ってみましょう。例題は

「I HATE PC9801M2. を表示するプログラムを書け」

です。

文字列の表示は自分で作ってもよいのですが，CP/M-86の BDOS コールに文字列のプリントがありますから，これを利用しましょう。文字列のプリントは

CL ： 09H

DX ： 文字列のオフセットアドレスをセット，

INT 224

とします。

プログラム全体はリスト3-3のような構成になります。

リスト3-3

```
        CSEG                            } プログラムのはじめはいつもこう書く
        ORG     100H
        MOV     CL,09H
        MOV     DX,OFFSET STRING
        INT     224                       文字列出力
                                          (プログラム本体)
        RETF

END_CS:
        DSEG                            } データのはじめはいつもこのように書く
        ORG     OFFSET END_CS
STRING  DB      'I HATE PC9801M2.$'       文字列
        END                               終わり
```

3節と4節が分かっていれば簡単です。

STRING　DB

'I HATE PC9801M2. $'

は

STRING DB 49H, 20H, 48H, 41H,

54H, 45H, 20H, 50H,

43H, 39H, 38H, 30H,

31H, 4DH, 32H, 2EH, 24H

と書いたのと同じことで，ASCII データをメモリに置くことを意味します。

文字列の最後が＄になっているのは，CP/M-86文字列のプリントに使う文字列の終わりが'$'で表されるためです。

プログラムの四角で囲まれている部分は，決まった型だと考えてください。四角で囲まれた部分以外はみなさんが自由にプログラムしてよい部分です。

このプログラムのファイル名をTEST2. A86とすると，実行には

ASM86　TEST2 ⏎

GENCMD　TEST2　8080 ⏎

TEST2 ⏎

とします。画面に

I HATE PC9801M2.

と表示されるはずです。

ASM86で注意する点

以上のほかに，ASM86で知っておいたほうがよいことがいくつかあります。

演算子

オフセット1234HにALレジスタの内容をストアしたいような場合に . 演算子を使います。これは

```
MOV    .1234H, AL
```

のようになります。

```
.1234H
```

は変数とみなされるわけです。『.』の後ろは数値でなければなりません。

8086ではベクトルを変更することがよくありますが，このとき . 演算子は役に立ちます。いま，COPYキーを無効にすることを考えます。COPYキーのベクトルは物理アドレスの14H〜17Hにオフセット，セグメントの順に入っています。これを082BH, FD80Hに変更します。アドレスFD80H:082BHはROMで，内容は

```
IRET
```

となっています。ですから，これでCOPYキーは無効になるはずです。プログラムは

```
XOR  AX, AX
MOV  ES, AX
CLI  注31
MOV  ES:WORD PTR .0014H, 082BH
MOV  ES:WORD PTR .0016H, 0FD80H
STI
```

となります。WORD PTR は次項を参照してください。

PTR演算子

SIレジスタ間接でメモリに12Hをストアすることを考えます。

```
MOV[SI], 12H
```

としたいところですが，12Hがバイトデータの12Hかワードデータの0012Hかアセンブラには分かりません。これを指定するのがPTR演算子です。つまり，

MOV BYTE PTR[SI], 12H

とすれば，BYTE PTR[SI] がバイト変数とみなされ，バイトデータの12Hがストアされますし，

MOV WORD PTR[SI], 12H

とすれば，WORD PTR[SI]がワード変数とみなされ，ワードデータの0012Hがストアされます。

数値定義

ASM86では2, 8, 10, 16進のデータが使えます。数値定数の最後は基数を表すB, O, D, H[注32]をつけます。

たとえば，

1010B　(2進数)
7701O　(8進数)
123D　(10進数)
0FFFFH　(16進数)

のように。また，10進数ではDを省略することができます。数値定数はラベル，変数と区別するため先頭は数でなければなりません。

FFFFH

は間違いとなります。16進データのときは気をつけてください。

以上で一応，ASM86によるプログラミングができるでしょう。ASM86のくわしい説明はこれから行います。

次に，少し長いASM86のプログラム例（リスト3-4）を示します。

リスト3-4

```
;
; ASM86 OHPC1
; GENCMD OHPC1 8080
; OHPC1
        CSEG                        } プログラムのはじめはこう書く
        ORG     100H
```

```
;1005 SCREEN 3,0
        MOV     AH,40H          } ROM BIOSを呼んで画面の表示を開始
        INT     18H

        MOV     AH,42H          | ROM BIOSを呼んで640×400
        MOV     CH,0C0H         | ドットモードに
        INT     18H

;1010 FOR H=-120 TO 120 STEP 15                   |
L1010:                                            |
        MOV     H,-120                            |
FOR1:                                             |
        CMP     H,120    ! JG    NEXT1            |
; 1020 GOSUB *DRAW.                               |
        CALL    DRAW                              |
; 1030 NEXT H                                     |
        ADD     H,15     ! JMP   FOR1             |
NEXT1:                                            | メインルーチン
; 1035 FOR H=115 TO -115 STEP -15                 |
        MOV     H,115                             |
FOR4:                                             |
        CMP     H,-115   ! JL    NEXT4            |
; 1037 GOSUB *DRAW.                               |
        CALL    DRAW                              |
; 1038 NEXT H                                     |
        SUB     H,15     ! JMP   FOR4             |
NEXT4:
; 1039 I$=INKEY$:IF I$="" THEN 1010
        MOV     AH,01H          } ROM BIOSを呼んでキー入力が
        INT     18H             } ないかを調べる
        CMP     BH,0
        JE      L1010
; 1040 END
        RETF  ---------------------------------- CCPへ
; 1050 *DRAW
DRAW:
; 1060 I=0
        MOV     I,0
; 1070 FOR X=0 TO 100 STEP 10
        MOV     X,0
FOR2:
        CMP     X,100    ! JG    NEXT2
; 1080  FOR Y=0 TO 100 STEP 10
        MOV     Y,0
FOR3:
        CMP     Y,100    ! JG    NEXT3
; 1090    Z=H*SIN((X-5)/90*3.14)*SIN((Y-5)/90*3.14)
        MOV     AX,X
        SUB     AX,5
        MOV     BX,180 ---------+
        IMUL    BX              |
        MOV     BX,90           |
        IDIV    BX              |
        CALL    SIN             +-sinルーチンがラジアンでなく「度」なので,
        PUSH    AX              |  πでなく180を使用
                                |
        MOV     AX,Y            |
        SUB     AX,5            |
        MOV     BX,180 ---------+
        IMUL    BX
        MOV     BX,90
        IDIV    BX
        CALL    SIN

        POP     CX
        IMUL    CX
```

```
        MOV     BX,100 ─┐
        IDIV    BX      │
                        ├─sinルーチンは実際のsin値の100倍を
        IMUL    H       │  返しているので，100で割る必要かある
                        │
        IDIV    BX ─────┘

        MOV     Z,AX
; 1100   C=ABS(Z)/120*6+1
        MOV     AX,Z      ┐
        CMP     AX,0      │
        JGE     L1        ├ ABS(AX)
        NEG     AX        ┘
L1:
        MOV     BX,6
        IMUL    BX

        MOV     BX,120
        IDIV    BX

        INC     AX
        MOV     C,AX

; 110   GOSUB *PSET.3D
        CALL    PSET_3D
; 1120 NEXT Y
        ADD     Y,10
        JMP     FOR3
NEXT3:
; 1130 NEXT X
        ADD     X,10
        JMP     FOR2
NEXT2:
; 1140 RETURN
        RET
; 1150 '
; 1160 *PSET.3D ──────────── 3次元の座標(x,y,z)を表示するルーチン
PSET_3D:
; 1170 PRESET(XE(I),YE(I))
        MOV     SI,I      ┐
        SHL     SI ,1     │
        MOV     AX,XE[SI] ├ 古い点を消す
        MOV     BX,YE[SI] │
        MOV     CX,0      │
        CALL    PSET      ┘
; 1180 XS=1.7320508#*(Y-X)+320
        MOV     AX,Y
        SUB     AX,X

        MOV     BX,17321
        IMUL    BX
        MOV     BX,10000
        IDIV    BX
        ADD     AX,320
        MOV     XS,AX

; 1190 YS=(X+Y)-Z+150
        MOV     AX,X
        ADD     AX,Y
        SUB     AX,Z
        ADD     AX,150
        MOV     YS,AX

; 1200 PSET(XS,YS),7
        MOV     AX,XS
        MOV     BX,YS
```

```
        MOV     CX,C
        CALL    PSET

; 1210 XE(I)=XS:YE(I)=YS:I=I+1
        MOV     SI,I
        SHL     SI,1
        MOV     AX,XS
        MOV     XE[SI],AX

        MOV     AX,YS
        MOV     YE[SI],AX

        INC     I
; 1220 RETURN
        RET
PSET:
; ARGUMENTS AX --> X-COORDINATES
;           BX --> Y-COORDINATES
;           CX --> COLOR CODE
        MOV     XX,AX
        MOV     YY,BX
        MOV     COLOR,CX

        MOV     SET,0
        TEST    COLOR,1
        JZ      P1
        MOV     SET,0FFFFH
P1:
        MOV     AX,0A800H
        MOV     ES,AX
        CALL    PSET0

        MOV     SET,0
        TEST    COLOR,2
        JZ      P2
        MOV     SET,0FFFFH
P2:
        MOV     AX,0B000H
        MOV     ES,AX
        CALL    PSET0

        MOV     SET,0
        TEST    COLOR,4
        JZ      P3
        MOV     SET,0FFFFH
P3:
        MOV     AX,0B800H
        MOV     ES,AX
        CALL    PSET0
        RET

PSET0:
        MOV     AX,XX
        SHR     AX,1
        SHR     AX,1
        SHR     AX,1
        MOV     SI,AX

        MOV     AX,YY
        MOV     BX,AX
        SHL     AX,1
        SHL     AX,1
        ADD     AX,BX

        SHL     AX,1
        SHL     AX,1
```

PSETルーチン
PSET(AX, BX), CX

```
        SHL     AX,1
        SHL     AX,1

        ADD     SI,AX
        MOV     AL,128
        MOV     CX,XX
        AND     CL,7

        SHR     AL,CL
        CMP     SET,0FFFFH
        JNZ     EPSET
        OR      ES:[SI],AL
        RET
EPSET:
        NOT     AL
        AND     ES:[SI],AL
        RET
SIN:
;
; SIN ROUTINE
;
;       ARGUMENT AX RETURN AX( 100*SIN(AX) )
;
        CMP     AX,0
        JL      SINM

        CWD
        MOV     BX,360
        IDIV    BX

        CMP     DX,90
        JG      SIN1                    ; IF DX>90 THEN SIN1

        SHL     DX,1                    ; DX=DX*2
        MOV     BX,DX

        MOV     AX,SIN_TABLE[BX]
        RET

SIN1:   CMP     DX,180
        JG      SIN2

        NEG     DX
        ADD     DX,180
        SHL     DX,1
        MOV     BX,DX
        MOV     AX,SIN_TABLE[BX]
        RET

SIN2:   CMP     DX,270
        JG      SIN3
        SUB     DX,180
        SHL     DX,1
        MOV     BX,DX
        MOV     AX,SIN_TABLE[BX]
        NEG     AX
        RET

SIN3:   NEG     DX
        ADD     DX,360
        SHL     DX,1
        MOV     BX,DX
        MOV     AX,SIN_TABLE[BX]
        NEG     AX
        RET
```

sinルーチン
100倍の値を返す

```
SINM:   NEG     AX
        CALL    SIN
        NEG     AX
        RET

END_CS:
        DSEG
        ORG     OFFSET END_CS                    } データのはじめにはこう書く
; 1000 DIM XE(120),YE(120)
XE      RW      121                              } 配列
YE      RW      121

H       DW      0
X       DW      0
Y       DW      0
Z       DW      0
I       DW      0

XS      DW      0                                  変数
YS      DW      0
C       DW      0

XX      DW      0
YY      DW      0
COLOR   DW      0
SET     DW      0

SIN_TABLE       DW      0,1,3,5,6,8,10,12,13,15
                DW      17,19,20,22,24,25,27,29,30,32
                DW      34,35,37,39,40,42,43,45,46,48
                DW      49,51,52,54,55,58,58,60,61,62
                DW      64,65,66,68,69,70,71,73,74,75
                DW      76,77,78,79,80,81,82,83,84,85     sinテーブル
                DW      86,87,88,89,89,90,91,92,92,93
                DW      93,94,95,95,96,96,97,97,97,98
                DW      98,98,99,99,99,99,99,99,99,99
                DW      100
        END
```

7 アセンブリ言語とコンパイラのスピード

16ビット用のソフトウェアの多くはコンパイラを使って作られています。確かに16ビット用コンパイラ，たとえばLattice Cなどのオブジェクトはかなり速いものです。また，開発効率もよいでしょう。しかし，コンパイラを使って作られたために

①オブジェクトが巨大

②あまり速くない

などの弊害が生じています。①の理由で128Kバイトの標準メモリで動作しないソフトウェアが数多くあります。MS－DOSやCP/M-86の高級言語はほとんどすべて128Kバイトでは動作しません。CP/M-80にも数多くの高級言語がありますが，これらは全部たった64Kバイトのメモリで動作しているのにです。

16ビットソフトウェアは処理が高度化されているのは事実です。16ビット用高級言語はたいていその言語のフルセット仕様となっています。しかし，必要メモリ量の増加を処理の高度化だけで説明することは無理でしょう。やはり，それはコンパイラが安直に使われたためだと思います。

また，②の期待した速度が得られないこともよく経験します。C言語で書かれたスクリーンエディタの多くは文字列サーチがかなり遅いようです。3000行くらいのテキストで先頭から最終行にある文字列をサーチすると，Word Master ならばすぐに見つけ出すのに，C言語によるスクリーンエディタでは数十秒もかかってしまいます。

また，多くのアセンブラ，言語プロセッサにはアセンブル/コンパイル速度がかなり遅いものもあります。もちろん，先に述べた処理の高度化のために遅くなっている部分もあるでしょう。しかし，それだけでは説明がつかないと思います。やはり，ここでもコンパイラが使われたがため，といわねばなりません。

16ビットCPUのパーソナルコンピュータを選んだ人の理由は，なんといっても「速い」ことだと思います。PC-9801は8801よりも機械語レベルで速いでしょう。一般に，16ビットCPUは8ビットCPUよりも速いのです。しかし，8ビットCPUのソフトウェアが機械語で作られ，16ビットCPUのソフトウ

ェアがコンパイラで書かれるならば，16ビットCPUを選んだメリットがないといえるのではないでしょうか。

コンパイラを使えば開発効率もよいでしょうし，保守もアセンブリ言語より容易でしょう。しかし，それは作る側の論理です。使う側としては開発が何年かかろうと，保守作業がどんなに大変だろうと知ったことではありません。使う側は速ければ速いほどよいはずです。

16ビットCPUは速く，また16ビットCPU用コンパイラのオブジェクトもけっこう速いものです。しかし，アセンブリ言語をいっさい使わなくてよいほど速いとは思えません。コンパイラのメーカーの出しているベンチマークをうのみにせず，コンパイラとアセンブリ言語で同一の実際に使うプログラムを書いてみてください。アセンブリ言語のほうが，かなり速い印象を受けると思います。

昨今，コンパイラをもてはやす記事が多く見られますが，それらはみな開発側からのものです。使う側としてはそのプログラムが何で書かれていようと関係ありません。速度の点を考えると，コンパイラでないほうがよいのかもしれません。

あるゲームのパッケージに

「このプログラムはコンパイラで書かれています」

と明示されたものがありますが，これはコンパイラのメーカーの宣伝にはなっても，ゲームメーカーの宣伝にはなっていないような気がします。

誤解してほしくないのですが，コンパイラを使うなといっているのではありません。BASICでは遅いがアセンブリ言語で書くのは面倒といったとき，コンパイラはとても便利なはずです。1本売っておしまいというプログラムがよくあります。そんなプログラムは，気合を入れてアセンブリ言語で書いてもしかたないでしょう。コンパイラで間に合うのなら，コンパイラを使って開発期間を短くしたほうがよいと思います。その結果，オブジェクトが巨大になっても，相手側のメモリを増設してもらえばすむことですから。

しかし，多くのユーザーを対象とした場合これでは困ります。全員がそのプログラムを実行できるメモリを積んでいるわけではありません。中には，VRAMにMASMをロードしてなんとかアセンブルしている人もいるのですから。

私のいいたいことは

「現在のパーソナルコンピュータはプログラムをすべてコンパイラで書けるほどの能力にはなっていない」

ということです。コンパイラで間に合う場合もあります。しかし，それはすべての場合ではありません。どうしてもアセンブリ言語が必要になる場合が出てきます。コンパイラを使っている人の多くが最初に読むマニュアルの項目が「アセンブリ言語とのインタフェース」であることもこれを示しています。パーソナルコンピュータはまだまだアセンブリ言語を必要としています。

PC-9801F3/M2などはメモリが256Kバイト標準となり，富士通のFM-16βは512Kバイト標準でCPUに80186（8MHz）を採用して速度が20〜30%アップするなどコンパイラを使いやすい環境となりつつありますが，アセンブリ言語を使う場面はまだまだなくなりそうもありません。

引き続きASM86の解説を行います。

アセンブラ擬似命令——(1)

アセンブリ言語には擬似命令があります。擬似命令は普通の命令と同じようにプログラム中に書かれ，アセンブラにさまざまな指示を与えます。ASM86にもORG，CSEG，IF～ENDIFなど25個のアセンブラ擬似命令とDBIT，RELB，MODRMなど9個のコードマクロ擬似命令があります。

コードマクロ擬似命令はあとで解説することにして，ここではアセンブラ擬似命令を解説します。

CODEMACRO

擬似命令CODEMACROはコードマクロ機能の使用開始を示します。コードマクロはあとでくわしく説明しますが，CPUの命令を作る機能です。たとえばNO_OPERATIONという命令を新しく作るには

```
CODEMACRO    NO_OPERATION
        DB    90H
ENDM
```

とします。こう宣言すると，プログラム中でNO_OPERATIONという命令が使えます。NO_OPERATIONはオブジェクトに90Hを生成します。

コードマクロ機能はこのような単純な命令からレジスタオペランド，メモリオペランドをもった複雑な命令まで作るようになっています。そのため，9個のコードマクロ擬似命令が用意されています。8086の全命令はコードマクロ機能ですべて記述することができます。また，NEC版CP/M-86には8087.LIBというファイルが入っていますが，これはコ・プロセッサ8087の命令用コードマクロライブラリです。つまり，8087の命令もコードマクロで作り出せるのです。

CSEG

CSEG擬似命令は以下がコードセグメントに属すことを示します。命令文はコードセグメントに置かなければなりませんから，命令文の先頭はCSEGで始めなければなりません。データは通常データセグメントに置きますが，コー

ドセグメントに置くこともできます。ただし，この場合はそのデータをアクセスしようとすると自動的に CS: が挿入されます。たとえば

```
        CSEG
        MOV   AX,   VAR
   VAR  DW    123
```

はオブジェクトレベルでは

```
   MOV    AX, CS:VAR
```

と書いたのと同じことになります。

CSEG には 3 つの形があります。

①CSEG　数値式

②CSEG

③CSEG　$

①はコードセグメントの位置が分かっているときに使います。たとえば8086を使った組み込みシステムでプログラムを ROM 化するような場合，つまり配置されるセグメントのアドレスが分かっている場合に使います。また，ゲームプログラムでブートローダを自分で書いて，勝手なアドレスにプログラムを置く場合にも使えます。

しかし，CP/M-86上で実行するプログラムを作るときには①は使えないことに注意してください。CP/M-86ではメモリ管理をCP/M-86自身が行います。CP/M-86は空き領域にプログラムをロードして実行しますが，空いているメモリ位置は動的に変化します。そのため，プログラム側がロードするセグメントアドレスを指定しても，そこが利用可能か分かりません。そのため，CP/M-86ではこのようなプログラムは

Memory not available

というエラーになります。①はCP/M-86以外の環境で実行するプログラムを作るときのみに使うとよいでしょう。したがって，CP/M-86で実行するプログラムには②，③を使います。②の CSEG は以下がコードセグメントに属すことを指定しますが，そのセグメントをどのセグメントアドレスに割りつけるかは指定しません。つまり，このプログラムは**リロケータブル**になります。8086はセグメントレジスタがあるためにリロケータブルなプログラムが作りやすい特徴があります。この場合のリロケータブルとは16バイト単位，つまりパラ

グラフ単位のリロケータブルということです。パラグラフ単位ならば，プログラムをどこに配置しても実行可能だということです。このため，CP/M-86上でも実行可能となるわけです。

③は中断されたコードセグメントを継続するのに使います。次のプログラムでは，

```
CSEG
MOV    AX, 0 ——コードセグメントに属す
DSEG
DB    4, 5, 6 ——データセグメントに属す
CSEG    $
RETF ————————コードセグメントに属す
```

とするとプログラムの表記上はコードセグメントが2つに分断されていますが，オブジェクトでは

```
MOV    AX, 0
RETF
```

と書いたのと同じ結果となります。

DB

擬似命令DB(DEFINE BYTE STRAGE) はバイト領域を初期化します。

```
DB    1,  2,  3
```

ではオブジェクトは

```
01H,  02H,  03H
```

となります。 DB では文字定数も使えます。

```
DB    'ABC'
```

とすればオブジェクトは

```
41H,  42H,  43H
```

となります。つまり，ASCII コードが置かれるのです。

DBが最も使われるのは変数の宣言としてでしょう。

```
X    DB    3
```

とすれば，初期値3をもったバイト変数Xが使えるようになります。

変数には4つの属性があります。

●セグメント属性
●オフセット属性
●型属性
●データ長
セグメント属性はその変数の属すセグメント値を示し，オフセット属性はオフセット値を，型属性はBYTE，WORD，DWORDを，データ長はバイト数を示します。これらを数値化する演算子に

SEG，OFFSET，TYPE，LENGTH

があります。

```
DSEG    1000H
ORG     2000H
STRING  DB
            `THIS IS A STRING.'
```

の場合

SEG STRING

の値は1000H

OFFSET STRING

の値は2000H

TYPE STRING

の値は1（つまり BYTE)

LENGTH STRING

の値は17となります。SEG演算子には注意が必要です。いまの例ではデータセグメントの位置が

DSEG 1000H

と指定してあったからよかったのですが,

```
DSEG
ORG    2000H
STRING DB ` THIS IS STRING.'
```

とすると

SEG STRING

は使えません。このプログラムはリロケータブルなため，データセグメントの

アドレスは実行時まで分からないのです。CP/M-86用のプログラムを作る場合，SEG演算子は使えないといえるでしょう。

ちなみに，MS-DOSのMASMのSEG演算子はMS-DOS用のプログラムを作る場合も使えます。

DD

擬似命令DDは4バイトの領域を初期化するときに使います。といっても，数値定数で初期化するのではなく，ファーコールやファージャンプを間接指定する際のテーブル作成に使います。

```
    JMPF    TABLE
              ⌇
    TABLE   DD    TEMP
```

では，TEMPというラベルにファージャンプします。

```
    DD    TEMP
```

の結果，TABLEにはTEMPのオフセットアドレス，セグメントアドレスが順々に格納されます。また，TABLEはDWORDという型属性をもちます。つまり，

```
    TYPE    TABLE  = 4
```

です。

TEMPのセグメント値が格納されると述べました。逆にいえば「セグメント値が決まっていなければDDは使えない」といえます。つまり，DDはCP/M-86で実行するプログラムでは使えないのです。CP/M-86以外の環境で実行するプログラムを書くためと考えてよいでしょう。

DSEG

擬似命令DSEGは以下がデータセグメントであることを示します。データの先頭にはDSEGを置くのが普通です。たとえば，次のように使います。

```
         DSEG
    X    DW    123
    Y    DB    10
```

DSEGはCSEGと同様3つの形式があります。

①DSEG　数値式

②DSEG

③DSEG　$

①はデータセグメントの配置されるアドレスが決まっているとき，つまり組み込みシステムのプログラムやゲームなどCP/M-86以外の環境で動作するプログラムに使います。CP/M-86上で動作するプログラムには使いません。

②は最もよく使う形式で，リロケータブルなプログラムを作るときに必要です。CP/M-86上で実行するプログラムはDSEGを使わなければなりません。

③は中断されたデータセグメントを継続するのに使います。

```
        DSEG
X       DW      0 ――データセグメントに属す
        CSEG    $
        RETF ――コードセグメントに属す
        DSEG    $
Y       DW          1 ――データセグメントに属す
```

ではデータセグメントが2つに分断されていますが，オブジェクトでは

```
X       DW      0
Y       DW      1
```

と書いたのと同じことになります。

DW

擬似命令DWはワード領域を初期化します。

```
    DW    1, 2, 3
```

では

```
   01, 00, 02, 00, 03, 00
```

というデータがオブジェクトとなります。

DWがよく使われるのはワード変数を宣言する場合でしょう。

```
X       DW      1234
```

は初期値1234をもったワード変数Xを宣言します。

また，間接ジャンプ用のジャンプテーブルを作ることもできます。DIレジスタの値によって

0 → TEST0

1 → TEST1

2 → TEST2

の各ルーチンにジャンプするプログラムは

```
        SHL    DI, 1
        JMP    TABLE  [DI]
                  ⟩
   TABLE DW  TEST0, TEST1, TEST2
```

となります。

EJECT

EJECT 擬似命令はリスティングファイルで改ページを指示します。オブジェクトには何の影響も与えません。リスティングを美しくするために使います。EJECT 自身は改ページ後に印字されます。

END

END はプログラムの終わりを示します。仮に END がなくても ASM86 はファイルの終わりとみなしますから，END はなくてもかまいません。

ENDIF

IF の項を参照してください。

ENDM

ENDM はコードマクロ定義の終わりを示します。コードマクロ機能はあとで解説します。

EQU

EQU はシンボルに値と属性を与えます。EQU には 4 つの種類があります。

①シンボル　EQU　数値式

②シンボル　EQU　アドレス式

③シンボル　EQU　レジスタ

④シンボル　EQU　命令ニーモニック

それぞれに例を示しましょう。

①の例

```
SUM EQU  1+2+3+4+5
```

で SUM は数値15と同じように使えます。

②の例

```
AET  EQU
ACTIVE__EDGE__TABLE
```

で，AET は ACTIVE__EDGE__TABLE と同じアドレスをもちます。

③の例

```
COUNT  EQU  CX
```

で，レジスタ CX に別の名前 COUNT を与えます。

④の例

```
NO__OPERATION EQU NOP
```

で，命令 NOP に別の名前 NO__OPERATION を与えます。

```
Stop EQU MOV
smoking EQU AX
boys EQU 1
```

とすることで

```
MOV AX, 1
```

と書く代わりに

```
Stop smoking, boys!
```

と書くこともできます。

ESEG

ESEG は以下がエクストラセグメントに属することを示します。データが64Kバイトを超える場合，このエクストラセグメントを使えば，128Kバイトまでは扱うことができます。

CP/M-86では，プログラムの規模に応じて

①8080モデル

②スモールモデル

③コンパクトモデル

が使えるようになっています。これまでは①の8080モデルしか扱ってきませんでしたが，これはコード，データ合わせて64Kバイト以内のプログラムしか作れません。

②のスモールモデルはコード64Kバイト，データ64Kバイトまでのプログラムが組めます。アセンブリ言語では，

```
CSEG
[コード]
DSEG
ORG     100H
[データ]
END
```

の形で書きます。

③のコンパクトモデルではコード64Kバイト，データ64Kバイトに加え，エクストラデータ64Kバイト，スタック領域64Kバイトまでのプログラムを書くことができます。

アセンブリ言語では

```
CSEG
[コード]
DSEG
ORG     100H
[データ]
ESEG
[エクストラデータ]
SSEG
[スタック領域]
END
```

と書きます。

コンパクトモデルまでしかサポートしなかったことは CP/M-86の欠点といえるかもしれません。プログラムによってはコードが64Kバイトに収まらなかったり，巨大な配列が必要でデータが64Kバイトを超えたりといったこともあ

りえます。しかし，CP/M-86では最も大きなコンパクトモデルでもこのようなプログラムを扱うことができません。特殊なローダを作ってこれを解決しているコンパイラもありますが，CP/M-86の標準的な機能としてもっているべきものではないでしょうか。

話を ESEG に戻しましょう。ESEG も CSEG，DSEG 同様 3 つの形式があります。

①ESEG　数値式

②ESEG

③ESEG　$

意味はそれぞれ CSEG，DSEG と同じです。

擬似命令の解説はこのくらいにして，アセンブリ言語によるプログラミング例に移ります。

ここでは SIN，PSET ルーチンを使って定常波のデモプログラムを作ってみました。

$$y_1=50\sin((x-vt)/640*8\pi)$$

$$y_2=50\sin((x+vt)/640*8\pi)$$

を合成すると

$$y=y_1+y_2$$

という定常波ができます。これを動かすプログラムです。

BASIC ではリスト3-5のようになるでしょうか。N_{88}-BASIC コンパイラにかけてみましたが，とてもリアルタイムと呼べるものにはなりませんでした。

リスト3-5

```
1000 DIM OY1(639),OY2(639),OY3(639)
1005 I=0
1007 PI=3.141592654#
1010 SCREEN 3,0
1020 WHILE INKEY$=""
1030 GOSUB *PLOT.SIN
1040 I=I+4:IF I>=160 THEN I=I-160
1050 WEND
1055 END
1060 *PLOT.SIN
1070 FOR X=0 TO 639 STEP 5
1080     Y1=50*SIN((X-1)*8*PI/640)
1090 Y2=50*SIN((X+I)*8*PI/640)
1100 Y3=Y1+Y2
1110  PSET(X,OY1(X)),0:PSET(X,OY2(X)),0:PSET(X,OY3(X)),0
```

```
1120  Y1=-Y1+200:Y2=-Y2+200:Y3=-Y3+200
1130   PSET(X,Y1),1:PSET(X,Y2),1:PSET(X,Y3),2
1140   OY1(X)=Y1:OY2(X)=Y2:OY3(X)=Y3
1150 NEXT X
1160 RETURN
```

アセンブリ言語の例をリスト3-6に示します。コメントにリスト3-5のステートメントを挿入しましたので，対応させながら読んでみてください。

リスト3-6

```
        CSEG
        ORG     100H

; 1007 PI=3.141592654#                     このプログラムでは角度の単位が「度」なので
PI      EQU     180                        π=180で計算
PLUS    EQU     0FFFFH
MINUS   EQU     NOT     PLUS
; 1010 SCREEN 3,0
        MOV     AH,40H                     ┐ 画面の表示を開始
        INT     18H                        ┘

        MOV     AH,42H                     ┐
        MOV     CH,0C0H                    │ 640×400ドットモードへ
        INT     18H                        ┘

; 1020 WHILE INKEY$=""
WHILE:
        MOV     AH,01H                     ┐
        INT     18H                        │ キー入力があればWENDへ
        OR      BH,BH                      │
        JNE     WEND                       ┘

; 1030 GOSUB *PLOT.SIN                     3つの曲線を表示
        CALL    PLOTSIN

; 1040 I=I+4:IF I>=160 THEN I=I-160
        ADD     I,4

        CMP     I,160   ! JL    L2
        SUB     I,160
L2:

; 1050 WEND
        JMPS    WHILE
WEND:

; 1055 END
        RETF ──────────────────────────── CP/M-86へ
; 1060 *PLOT.SIN
PLOTSIN:

; 1070 FOR X=0 TO 639 STEP 5
        MOV     X,0
FOR:
        CMP     X,639   ! JLE   L1
        JMP     NEXT
L1:
```

```
; 1080  Y1=50*SIN((X-1)*8*PI/640)
        MOV     AX,X     ! SUB    AX,I
        MOV     BX,8*PI  ! IMUL   BX
        MOV     BX,640   ! IDIV   BX
        CALL    SIN

        CWD
        MOV     BX,20    ! IDIV   BX        } 三角関数は1000倍の値を返すので,
        MOV     Y1,AX                       } 20で割るとちょうどよくなる

; 1090 Y2=50*SIN((X+I)*8*PI/640)
        MOV     AX,X     ! ADD    AX,I
        MOV     BX,8*PI  ! IMUL   BX
        MOV     BX,640   ! IDIV   BX
        CALL    SIN

        CWD
        MOV     BX,20    ! IDIV   BX
        MOV     Y2,AX

; 1100 Y3=Y1+Y2
        MOV     AX,Y1    ! ADD    AX,Y2
        MOV     Y3,AX

; 1110  PSET(X,OY1(X)),0:PSET(X,OY2(X)),0:PSET(X,OY3(X)),0
        MOV     AX,X
        MOV     SI,AX    ! SHL    SI,1
        MOV     BX,OY1[SI]
        MOV     CX,0
        PUSH    SI
        CALL    PSET
        POP     SI

        MOV     AX,X
        MOV     BX,OY2[SI]
        MOV     CX,0
        PUSH    SI
        CALL    PSET
        POP     SI

        MOV     AX,X
        MOV     BX,OY3[SI]
        MOV     CX,0
        PUSH    SI
        CALL    PSET
; 1120  Y1=-Y1+200:Y2=-Y2+200:Y3=-Y3+200
        NEG     Y1       ! ADD    Y1,200
        NEG     Y2       ! ADD    Y2,200
        NEG     Y3       ! ADD    Y3,200

; 1130   PSET(X,Y1),1:PSET(X,Y2),1:PSET(X,Y3),2
        MOV     AX,X
        MOV     BX,Y1
        MOV     CX,1
        CALL    PSET

        MOV     AX,X
        MOV     BX,Y2
        MOV     CX,1
        CALL    PSET

        MOV     AX,X
        MOV     BX,Y3
        MOV     CX,2
        CALL    PSET
```

```
; 1140   OY1(X)=Y1:OY2(X)=Y2:OY3(X)=Y3
        POP     SI
        MOV     AX,Y1    ! MOV    OY1[SI],AX
        MOV     AX,Y2    ! MOV    OY2[SI],AX
        MOV     AX,Y3    ! MOV    OY3[SI],AX

; 1150 NEXT X
        ADD     X,5      ! JMP    FOR
NEXT:

; 1160 RETURN
        RET

PSET:
; WRITE OE PIXEL
;
; ARGUMENTS:
;       AX X-COORDINATES BX Y-COORDINATES CX COLOR CODE
;
        MOV     XX,AX
        MOV     YY,BX
        MOV     C,CX

        MOV     W,0
        TEST    C,1
        JZ      P1
        MOV     W,0FFFFH
P1:
        MOV     AX,0A800H
        MOV     ES,AX
        CALL    PW0


        MOV     W,0
        TEST    C,2
        JZ      P2
        MOV     W,0FFFFH
P2:
        MOV     AX,0B000H
        MOV     ES,AX
        CALL    PW0

        MOV     W,0
        TEST    C,4
        JZ      P3
        MOV     W,0FFFFH
P3:
        MOV     AX,0B800H
        MOV     ES,AX
        CALL    PW0
        RET
PW0:
        MOV     AX,XX
        SHR     AX,1
        SHR     AX,1
        SHR     AX,1
        MOV     SI,AX

        MOV     AX,YY
        MOV     BX,AX
        SHL     AX,1
        SHL     AX,1
        ADD     AX,BX

        SHL     AX,1
        SHL     AX,1
```

点を打つルーチン

```
        SHL     AX,1
        SHL     AX,1

        ADD     SI,AX
        MOV     AL,128
        MOV     CX,XX
        AND     CL,7

        SHR     AL,CL
        CMP     W,0FFFFH
        JNZ     EPW
        OR      ES:[SI],AL
        RET
EPW:
        NOT     AL
        AND     ES:[SI],AL
        RET

SIN:
;
; SIN FUNCTION
;
; ARGUMENT:  AX ( DEGREE )
; RETURN:    AX (1000*SIN)
;
        CWD
        MOV     BX,360
        IDIV    BX

        CMP     DX,0
        JL      SINM

        MOV     BP,PLUS
        JMPS    SIN0
SINM:
        MOV     BP,MINUS
        NEG     DX
SIN0:
        CMP     DX,90
        JG      SIN1

        SHL     DX,1
        MOV     BX,DX

        MOV     AX,SINTABLE[BX]
        JMP     SINRET

SIN1:   CMP     DX,180
        JG      SIN2
        NEG     DX
        ADD     DX,180
        SHL     DX,1
        MOV     BX,DX
        MOV     AX,SINTABLE[BX]
        JMP     SINRET

SIN2:   CMP     DX,270
        JG      SIN3

        SUB     DX,180
        SHL     DX,1
        MOV     BX,DX
        MOV     AX,SINTABLE[BX]
        NEG     AX
        JMP     SINRET
```

sinを求めるルーチン

```
SIN3:   NEG     DX
        ADD     DX,360
        SHL     DX,1
        MOV     BX,DX
        MOV     AX,SINTABLE[BX]
        NEG     AX
SINRET:
        CMP     BP,MINUS
        JE      SINMI
        RET
SINMI:
        NEG     AX
        RET

ENDCS:
        DSEG
        ORG     OFFSET ENDCS
; 1000 DIM OY1(639),OY2(639),OY3(639)
OY1     RW      639+1
OY2     RW      639+1
OY3     RW      639+1

; 1005 I=0
I       DW      0

X       DW      0
Y1      DW      0
Y2      DW      0
Y3      DW      0

XX      DW      0
YY      DW      0
C       DW      0
W       DW      0

SINTABLE        DW      0 , 17 , 34 , 52 , 69 , 87 , 104 , 121
                DW      139 , 156 , 173 , 190 , 207 , 224 , 241 , 258
                DW      275 , 292 , 309 , 325 , 342 , 358 , 374 , 390
                DW      406 , 422 , 438 , 453 , 469 , 484 , 499 , 515
                DW      529 , 544 , 559 , 573 , 587 , 601 , 615 , 629
                DW      642 , 656 , 669 , 681 , 694 , 707 , 719 , 731
                DW      743 , 754 , 766 , 777 , 788 , 798 , 809 , 819
                DW      829 , 838 , 848 , 857 , 866 , 874 , 882 , 891
                DW      898 , 906 , 913 , 920 , 927 , 933 , 939 , 945
                DW      951 , 956 , 961 , 965 , 970 , 974 , 978 , 981
                DW      984 , 987 , 990 , 992 , 994 , 996 , 997 , 998
                DW      999 , 1000,1000

        END
```

1000倍のサインテーブル

アセンブラ擬似命令——(2)

IF

擬似命令 IF は ENDIF までの行をアセンブルしたりしなかったりすることができます。IF を使えば，違った機械でも動作するプログラムを書くことができます。

たとえば PC–9801 と同 F ではクロックが違うので LOOP 命令でウエイトすると，ウエイト時間が変わってしまいます。しかし，IF を使えば，

```
PC9801    EQU   -1
          IF   PC9801
          MOV   CX, 5000
          ENDIF
          IF   NOT   PC9801
          MOV   CX, 8000
          ENDIF
```

とすることによって，9801，F のどちらにも対応するプログラムが書けます。上の例では，9801用になっています。F 用にするには EQU の行を

PC9801　EQU 0 ——————————①

に変えるだけですみます。

IF の後ろには式がくる約束になっており，式が 0 ならばアセンブルせず，0 以外ならアセンブルします。いまの例では

```
IF     PC9801
IF     NOT   PC9801
```

などを使いましたが，比較演算子なども使えます。

```
IF     X   GE   300    (Xが300以上)
IF     USER   EQ 3  (ユーザーが3人)
              ⋮
```

IF で大切なことは，**アセンブル時に** IF の引数が評価されることです。実行時に評価されるわけではありません。最初の例で9801用としてアセンブルされ

たオブジェクトは9801用でしかありません。Fで実行するには①として，再アセンブル時にすでに決まっていなければなりません。IFのネストは5つまで可能です。

```
  IF   PC9801F
  IF   CLOCK8
    CALL   WAIT 8
  ENDIF
  IF   CLOCK 5
    CALL   WAIT 5
  ENDIF
ENDIF
```

なども正確にアセンブルされます。

IFはほんの少しの変更でほかの機械でも動くプログラムが作れるといった利点があります。なにしろ，2つ以上のプログラムを単一のソースプログラムで作れるのですから。

また，デバッグ時にしか使わない部分を

```
  IF   DEBUG
        :
  ENDIF
```

とすれば，デバッグ後ソースを変更したりデバッグ用のプログラムを別にもつ必要もなく，便利かもしれません。

INCLUDE

擬似命令INCLUDEはほかのファイルをプログラム中に展開します。

BEEP. LIBというファイルの内容が

```
BEEP:MOV   AL,  6
     OUT   37H,  AL
     MOV   CX,  10000
     LOOP  B
     MOV   AL,  7
     OUT   37H,  AL
```

```
        RET
```
だったとします。このサブルーチンBEEPを別のプログラムで使いたい場合

```
    CALL   BEEP
         :
    INCLUDE   BEEP.LIB
```
と書けばよいのです。

INCLUDEで指定するファイルのエクステンションは省略できます。その場合．A86だということになります。ドライブ名も省略でき，そのときはソースファイルのドライブとみなされます。

INCLUDEはネストできません。つまり，展開されるファイルはINCLUDE命令を含んではならないのです。

INCLUDEはよく使うサブルーチンをライブラリ化することができて便利ですが，多用すると処理が遅くなります。

LIST, NOLIST

LIST，NOLIST命令はリスティングファイルへの出力を制御します。

```
    NOLIST
```
とすれば，以後リスティングファイルへの出力は停止します。

```
    LIST
```
とすれば，ふたたび出力が始まります。

これらはデバッグ時などに便利でしょう。プログラムのデバッグ時にはここからここまではデバッグが終わっていて，ここからここまではデバッグ中という状態になります。そんなとき，すべての行を出力するのは時間の無駄です。

```
    LIST
    デバッグ中のルーチン
    NOLIST
    デバッグの終わったルーチン
```
とすれば時間の短縮になります。

ORG

擬似命令ORGはロケーションカウンタの値を変更します。アセンブラは現

在のコードやデータをどのアドレスに割りつけるかをロケーションカウンタとしてもっています。通常は逐次増えるようになっていますが，ORG で変更することができます。

CP/M-86ではベースページを確保するのに

```
ORG   100H
```

をよく使います。また，8080モデルで組む場合，

```
END__CS:
          DSEG
          ORG   OFFSET   END__CS
```

とすることもご存じでしょう。

ちなみに ORG は ORIGIN の略です。

PAGE WIDTH

擬似命令 PAGEWIDTH はリスティングファイルの横幅(カラム数)を指定します。プリンタのカラム数や画面のカラム数に合わせて変更する必要があります。デフォルト値は120で，画面に出力する場合は79です。

RB

擬似命令 RB は初期化されないバイト領域を確保します。

```
X      RB      10
```

とすれば，10バイトの初期化されないメモリが確保されます。このとき，X の型はバイトになります。RB は配列を宣言するのに使えます。

```
DIM   A (100)
```

は

```
A      RB      100+1
```

となります。

RB は Reserve Byte strage の略だと思われます。

RS

擬似命令 RS は初期化されないメモリを確保します。シンボルにはバイト属性を与えません。

X　　RS　　10

とすれば，10バイトのエリアが確保されますが，シンボルXの型はバイトになりません。

RW

擬似命令 RW は初期化されないワード領域が確保されます。

バイトがワードになっただけで，ほかは RB とまったく同じです。

X　　RW　　10

とすれば，10ワードのメモリが確保されます（10バイトではありません）。またXの型はワードとなります。

SIMFORM

擬似命令 SIMFORM はフォームフィード文字を対応するラインフィードと置き換えます。これはフォームフィード機能のないプリンタでリスティングファイルを打ち出すのに使います。

たいていのプリンタはフォームフィードがついているでしょうから，SIMFORMを使うことはないでしょう。

SSEG

擬似命令 SSEG はスタックセグメントの開始を宣言します。CSEG，DSEG，ESEG と同様 SSEG も

①SSEG　数値式

②SSEG

③SSEG　$

が使えます。①はスタックセグメントの位置が決まっているときに使います。

SSEG　　1234H

とすれば，以下のスタックセグメントは物理アドレスの12340Hから始まっていることになります。

①を使ったプログラムはリロケータブルではないので，CP/M-86上で実行するわけにはいかなくなります。

②は以下がスタックセグメントであることを示しますが，そのアドレスは指

定していません。CP/M-86上で実行するなど，リロケータブルなコードを生成する必要がある場合はSSEGを使います。

③は以前に中断されたスタックセグメントの続きから始めることを示します。

最初のうちは8080モデルで組むことが多いのでSSEGを使うことはないでしょう。

TITLE

TITLE擬似命令はリスティングファイルの各ページの先頭にタイトルを出力します。

TITLE `Yet Another Compiler Compiler'

とすれば，各ページの頭に

Yet Another Compiler Compiler

と表示されます。

10 演算子

これまでOFFSET, PTRなどの演算子がいくつか出てきました。ここでASM86のすべての演算子をまとめてみましょう。

●論理演算子

XOR, OR, AND, NOT

●比較演算子

EQ, LT, LE, GT, GE, NE

●算術演算子

+, −, *, /, MOD, SHL, SHR,

●セグメントオーバーライド

CS:, DS:, ES:, SS:

●その他

SEG, OFFSET, TYPE, LENGTH,

LAST, PTR, . ,$

論理演算子

論理演算子はブール代数の論理演算を行います。BASICにもあるので分かると思います。

```
0011B  XOR  0101B→0110B
0011B  OR   0101B→0111B
0011B  AND  0101B→0001B
NOT   0011B  →  1100B
```

8086のオペレーションコードにも同じ名前のものがあるので注意してください。

以下も，もちろん正しい例です。

```
XOR   AL,  X  XOR  Y
OR    AL,  X   OR  Y
AND   AL,  X  AND  Y
```

比較演算子

比較演算子は比較の結果，真(0FFFFH)や偽(0000H)を返します。擬似命令IF の引数に使われることが多いと思います。比較は**符号なし**で行われます。

```
X  EQ  Y  →  X=Y?
X  LT  Y  →  X<Y?
X  LE  Y  →  X≦Y?
X  GT  Y  →  X>Y?
X  GE  Y  →  X≧Y?
X  NE  Y  →  X≠Y?
```

FORTRAN のようで気持ちが悪いですね。

例を示します。

```
IF  CLOCK  EQ  5
    MOV  CX, 5000
ENDIF
IF  CLOCK  GE  8
    MOV  CX, 8000
ENDIF
```

算術演算子

算術演算子は和差積商などを返します。

```
X+Y      和
X-Y      差
(Xはラベル，変数，数値，Yは数値)
X  *  Y  積
X  /  Y  商
X MOD Y  余り
X SHL Y  左シフト
X SHR Y  右シフト
+X       X
-X       0-X
```

（X，Yは数値）

X，Yは**符号なし**数値であることに注意してください。

セグメントオーバーライド

CS:，DS:，ES:，SS:

はよくお分かりでしょう。ほかの演算子とともに使われるとき，注意してください。

MOV WORD PTR. 0002H，0FD80H

にCS: を入れるとしたら，

MOV CS:WORD PTR.
0002H，0FD80H

です。　MOV WORD PTR CS:
0002H，0FD80H

ではありません。

その他

SEG，OFFSET，TYPE，PTR,. などはもう説明しましたが，ざっとまとめておきましょう。

SEG　X

はXの含まれるセグメントのパラグラフアドレスを返します。

```
    DSEG    1234H──────────①
X   RW         1
```

のとき，SEG X は1234H となります。CP/M-86上で実行するプログラムはリロケータブルに作らなければならないため，①のようにアドレス指定をすることはありません。そのため，SEG演算子はほとんど使わないでしょう。なおSEG の引数はラベルか変数です。

OFFSET　X

はXのオフセット値を返します。8080モデルの決まり文句

```
END_CS:
DSEG
ORG OFFSET END_CS
```

でおなじみでしょう。また，ブロック転送のときにもよく使います。

```
DS:X
```

から100バイトを

```
ES:Y
```

以降に転送するには

```
CLD
MOV SI, OFFSET Y
MOV DI, OFFSET Y
MOV CX, 100
REP STOSB
```

とします。

TYPE 演算子は変数やラベルの型を返します。

たとえば，

```
X    RB    1
Y    RW    1
```

ならば

```
TYPE X=1
TYPE Y=2
```

となります。

LENGTH 演算子は変数の大きさ(バイト単位)を返します。たとえば，

```
X    DB `ABCDEFG´
```

とすれば

```
LENGTH X
```

は7になります。

これもブロック転送に使えます。

```
X    DB   `THIS IS STRING´,0
          (データセグメントにあるとする)
BUFFER    RB    100
     (エクストラセグメントにあるとする)
```

のときXの内容をバッファにコピーするには

```
CLD
```

```
MOV SI, OFFSET X
MOV DI, OFFSET BUFFER
MOV CX, LENGTH X
REP STOSB
```

と書きます。

LAST 演算子は

LENGTH−1

を返します。

LENGTH が 0 ならば 0 になります。

```
X    DB   0, 1, 2, 3, 4
          ─────────────↑__LAST Xはここが0から数えて何番目であるかを示す
          LENGTH Xは全部
          で何個あるかを示す
```

のとき

LENGTH X=5

LAST X=4

となります。

PTR 演算子は仮想変数やラベルを作るのに使います。

たとえば，バイトの34Hを[SI]に入れるとすれば，

```
MOV[SI], 34H
```

ではエラーになります。[SI]がバイトなのかワードなのか分からないためです。これを避けるには

```
MOV    BYTE PTR [SI], 34H
```

と PTR を使ってバイトであることを示す必要があります。

8086ではリセット時に

FFFFH:0000H

にジャンプしますが，プログラムでここに飛ぶためには，

```
JMP CS:DWORD PTR RESET
RESET:DW 0000H, OFFFFH
```

とします。

演算子は数値から変数を作り，オフセットアドレスの1000Hに12Hを入れるには

MOV BYTE PTR . 1000H, 12H

とします。8ビット系CPU用のアセンブラでは

[1000H]

と表記していたのが, . 1000Hとなったと考えればよいでしょう。

S演算子は現在のロケーションカウンタをオフセットにもつラベルです。だから,

JMP　S

は

HERE:　　JMP HERE

と同じような働きをします（無限ループ）。

Sはウエイトのときなどに

MOV CX, 1234H

LOOP S

などと使うことがよくあります。

以上で ASM86の擬似命令と演算子の解説は終わります。

例によって ASM86のソースプログラムの例をリスト3-7に示します。

リスト3-7

```
TRUE    EQU     0FFFFH
FALSE   EQU     NOT TRUE

        ORG     100H
        MOV     AH,40H          ! INT   18H
        MOV     AH,42H          ! MOV   CH,0C0H
        INT     18H
DEMO:
        MOV     AH,01H
        INT     18H
        CMP     BH,0
        JNE     DEMO1

        MOV     AX,400          ! CALL  RND     ! ADD   AX,100  ! MOV   X1,AX
        MOV     AX,400          ! CALL  RND     ! ADD   AX,100  ! MOV   X2,AX
        MOV     AX,200          ! CALL  RND     ! ADD   AX,100  ! MOV   Y1,AX
        MOV     AX,200          ! CALL  RND     ! ADD   AX,100  ! MOV   Y2,AX
        MOV     AX,8            ! CALL  RND     ! MOV   COLOR,AX
        CALL    BOX

        JMPS    DEMO
DEMO1:
        RETF

BOX:
;
; X1,Y1,X2,Y2,COLOR
;
        MOV     AX,X1
        CMP     AX,X2           ! JLE   B1
```

```
        XCHG    AX,X2           ! XCHG  AX,X1
B1:
        MOV     AX,Y1
        CMP     AX,Y2           ! JLE   B2
        XCHG    AX,Y2           ! XCHG   X,Y2
B2:
        MOV     AX,Y1           ! MOV   Y,AX
BOX_LOOP:
        MOV     AX,Y
        CMP     AX,Y2           ! JG    B3

        CALL    PLINE

        INC     Y               ! JMPS  BOX_LOOP
B3:
        RET

PLINE:
        MOV     SI,0
        TEST    COLOR,1
        JZ      P1
        MOV     DX,TRUE         ! CALL  PLIN    ! JMPS  P2
P1:
        MOV     DX,FALSE        ! CALL  PLIN
P2:
        MOV     SI,1
        TEST    COLOR,2
        JZ      P3
        MOV     DX,TRUE         ! CALL  PLIN    ! JMPS  P4
P3:
        MOV     DX,FALSE        ! CALL  PLIN
P4:
        MOV     SI,2
        TEST    COLOR,4
        JZ      P5
        MOV     DX,TRUE         ! CALL  PLIN    ! JMPS  P6
P5:
        MOV     DX,FALSE        ! CALL  PLIN
P6:
        RET

PLIN:
        MOV     AX,X1   ! MOV   PLINE_X1,AX
        MOV     BX,X2   ! MOV   PLINE_X2,BX
        MOV     CX,Y    ! MOV   PLINE_Y,CX
        MOV     PLINE_PSET,DX
        MOV     PLINE_COLOR,SI

        CLD
        CMP     PLINE_COLOR,0
        JNE     PLINEC1
        MOV     AX,0A800H
        JMPS    PLINEC2
PLINEC1:
        CMP     PLINE_COLOR,1
        JNE     PLINEC3
        MOV     AX,0B000H
        JMPS    PLINEC2
PLINEC3:
        MOV     AX,0B800H
PLINEC2:
        MOV     ES,AX                           ; ES=SEGMENT OF VRAM

        MOV     AX,PLINE_X1
        CMP     AX,PLINE_X2
        JLE     PLINE1
        XCHG    AX,PLINE_X2                     ; SWAP PLINE_X1, PLINE_X2
        XCHG    AX,PLINE_X1
PLINE1:
        MOV     AX,PLINE_X1
        MOV     CL,4
        SHR     AX,CL
```

BOXルーチン
水平方向のラインルーチンPLINを呼んで箱を描く
引数X1, Y1, X2, Y2, COLORはⒶによって乱数で決定

(X1, Y1)
(X2, Y2)

以下，水平な直線を描くルーチン
引数は
X1
X2
Y
DX(描画か消去かの指定)
SI（描画，プレーン0, 1, 2の指定）

```
        SHL     AX,1                                : PLINE_XE1=(PLINE_X1¥16)*2
        MOV     PLINE_XE1,AX

        MOV     AX,PLINE_X2
        SHR     AX,CL
        SHL     AX,1
        MOV     PLINE_XE2,AX                        ; PLINE_XE2=(PLINE_X2¥16)*2

        MOV     DI,PLINE_Y
        MOV     BX,DI
        SHL     DI,1
        SHL     DI,1
        ADD     DI,BX                   ; DI=5*PLINE_Y

        SHL     DI,1
        SHL     DI,1
        SHL     DI,1
        SHL     DI,1                    ; DI=16*5*PLINE_Y

        ADD     DI,PLINE_XE1
        MOV     AD,DI

        MOV     BX,PLINE_X1
        AND     BX,000FH

        CMP     BX,0
        JNE     PLINE2
        MOV     PLINE_DI,0FFFFH
        JMP     PLINE3
PLINE2:
        DEC     BX
        SHL     BX,1
        MOV     AX,OFFSET FILL_PATTERN[BX]
        NOT     AX
        MOV     PLINE_DI,AX
PLINE3:
        MOV     BX,PLINE_X2
        AND     BX,000FH
        SHL     BX,1
        MOV     AX,OFFSET FILL_PATTERN[BX]
        MOV     PLINE_D2,AX

        MOV     AX,PLINE_XE1
        CMP     AX,PLINE_XE2
        JE      PLINE_CASE1
        JMP     PLINE_CASE2
PLINE_CASE1:
;
        MOV     AX,PLINE_D1
        AND     AX,PLINE_D2
        MOV     DA,AX

        MOV     AX,DA
        MOV     DI,AD

        CMP     PLINE_PSET,TRUE
        JNE     PLINE_CASE11
        OR      ES:WORD PTR[DI],AX
        RET
PLINE_CASE11:
        NOT     AX
        AND     ES:WORD PTR[DI],AX
        RET

PLINE_CASE2:
        MOV     AX,PLINE_D1

        MOV     DI,AD

        CMP     PLINE_PSET,TRUE
        JNE     PLINE_CASE22
        OR      ES:WORD PTR [DI],AX
```

```
        JMP     PLINE_CASE23
PLINE_CASE22:
        NOT     AX
        AND     ES:WORD PTR [DI],AX
PLINE_CASE23:

        ADD     AD,2

        MOV     DI,AD                       ; ADDRESS

        MOV     CX,PLINE_XE2
        SUB     CX,PLINE_XE1
        DEC     CX
        DEC     CX

        SHR     CX,1                        ; CX = WORD NUMBER

        CMP     PLINE_PSET,TRUE
        JNE     PLINE_CASE24

        MOV     AX,0FFFFH
        REP     STOSW
        MOV     AX,PLINE_D2
        OR      ES:[DI],AX
        RET
PLINE_CASE24:
        XOR     AX,AX
        REP     STOSW
        MOV     AX,PLINE_D2
        NOT     AX
        AND     ES:[DI],AX

        RET
RND:
        MOV     CX,AX                       ; SAVE AX  ┐
        MOV     AX,259                                 │
        MUL     SEED                                   │
        ADD     AX,3                                   │
        AND     AX,32767                               ├ 乱数ルーチン
        MOV     SEED,AX                                │
        MUL     CX                                     │
        MOV     BX,32767                               │
        DIV     BX                                     │
        RET                                            ┘

END_CS:
        DSEG
        ORG     OFFSET END_CS

X1      DW      0                                          ┐
X2      DW      0                                          │
Y1      DW      0                                          │
Y2      DW      0                                          │
                                                           │
Y       DW      0                                          │
COLOR   DW      0                                          │
COUNT   DW      0                                          │
                                                           │
PLINE_X1        DW      0                                  │
PLINE_X2        DW      0                                  │
PLINE_y         DW      0                                  │
PLINE_COLOR     DW      0                                  │
                                                           │
PLINE_D1        DW      0                                  │
PLINE_D2        DW      0                                  │
                                                           │
AD      DW      0                                          ├ 変数
DA      DW      0                                          │
                                                           │
PLINE_XE1       DW      0                                  │
PLINE_XE2       DW      0                                  ┘
```

```
PLINE_PSET          DW        TRUE

FILL_PATTERN        EQU       $
            DW      00080H,000C0H,000E0H,000F0H,000F8H,000FCH,000FEH,000FFH
            DW      080FFH,0C0FFH,0E0FFH,0F0FFH,0F8FFH,0FCFFH,0FEFFH,0FFFFH

SEED        DW      1234
            END
```

11 コードマクロ機能

ここではASM86のコードマクロ機能を説明します。コードマクロ機能を簡単にいえば

「新しいオペコードを作る機能」

といえます。ASM86は8086のすべてのオペコードを受けつけますが，そのほかに8087のオペコードや自分で作ったオペコードもアセンブルできるようにするのがコードマクロ機能です。CP/M-86のマニュアルが不親切なため，この機能を使っている人はあまりないようです。そこで，コードマクロ機能の簡単な使い方を解説することにします。

最も簡単な例

```
MOV AL, 6 ──────────── ①
```

に相当する新しいオペコードMOV6を作ることを考えてみましょう。

先に答えを書きます。

```
CODEMACRO    MOV6
        DB    0B0H ┐
                   ├ ①のオブジェクトコード
        DB    06H  ┘
ENDM
```

上記のように宣言しておけば，プログラム中でMOV6をあたかも8086のオペコードのように使うことができます。たとえば次のようにです。

```
CSEG
MOV6
OUT 37H, AL
```

MOV6のように引数のない場合，新しいオペコードを作るのにはいたって簡単で，上に示したように

```
CODEMACRO    オペコード名
```

で宣言を開始し，次に実際のオブジェクトコードを

```
DB    0B0H
DB    06H
```

などとDB擬似命令を使って列挙し，

```
    ENDM
```

で宣言を終了したことを示します。

DBは通常の擬似命令にもありましたが，この場合はコードマクロ用の擬似命令で別物と考えてください。そのため，これまで許された

```
    DB      0B0H, 06H
```

のように並べて記述する方法はコードマクロ擬似命令であるDBでは許されず，

```
    DB      0B0H
    DB      06H
```

のように1つひとつ区切って書かなければなりません。

いまの例は引数がない場合ですが，これだけでもかなり使えます。

- ●全レジスタを退避する命令（PUSHALL）
- ●全レジスタを復帰する命令（POPALL）
- ●プログラムを終了する命令（END_OF_PROGRAM）

などは簡単に作れることが分かるでしょう。この3つは次のようになります。

```
    CODEMACRO PUSHALL
            DB      50H  ;PUSH AX
            DB      53H  ;PUSH BX
            DB      51H  ;PUSH CX
            DB      52H  ;PUSH DX
            DB      56H  ;PUSH SI
            DB      57H  ;PUSH DI
            DB      55H  ;PUSH BP
            DB      1EH  ;PUSH DS
            DB      06H  ;PUSH ES
    ENDM
```

同様に

```
    CODEMACRO POPALL
            DB      07H  ;POP ES
            DB      1FH  ;POP DS
```

```
        DB    5DH   ;POP BP
        DB    5FH   ;POP DI
        DB    5EH   ;POP SI
        DB    5AH   ;POP DX
        DB    59H   ;POP CX
        DB    5BH   ;POP BX
        DB    58H   ;POP AX
    ENDM
```

プログラムの終了は

```
    CODEMACRO END_OF_PROGRAM
        DB  30H    ;  } XOR CL, CL
        DB  0C9H   ;  }
        DB  30H    ;  } XOR DL, DL
        DB  0D2H   ;  }
        DB  0CDH   ;  } INT 224
        DB  0E0H   ;  }
    ENDM
```

これらは単なる例だけでなく，実際に役に立つコードマクロだと思います。インタラプトを使ったプログラムでは全レジスタを退避したり復帰したりすることが多いのですが，コードマクロ機能がない場合，毎回

```
    PUSH  AX
    PUSH  BX
        ⋮
```

や

```
    POP  ES
    POP  PS
        ⋮
```

と書かなければならず，とても面倒なものですが，先のようにコードマクロ定義しておけば単に

```
    PUSHALL
```

や

POPALL

と書くだけですんで，手間がはぶけるだけでなく，退避や復帰する順番を書き間違えることもなくなります。

引数のある場合

引数のない場合は前項のように簡単ですが，引数のある場合は少し複雑になります。

イミディエイトの引数

MOV　AL，6

のように

MOV　AL

に相当するオペコード MOVAL1を作ることを考えましょう。ただし，引数は数値に限り，

MOVAL1 6

のように使うこととします。

これも答えを先に書きます。

```
CODEMACRO    MOVAL1
          PAR : DB――――――②
        DB      0B0H
        DB      PAR
ENDM
```

ここで，

MOV AL，0 → 0B0H，00H
MOV AL，1 → 0B0H，01H
MOV AL，2 → 0B0H，02H

ですから

MOV AL，PAR

のオブジェクトコードは 0B0H，PAR となります。

②の部分はこれから定義しようとするオペコードが MOVAL1 であり，引数を PAR で表しその引数がイミディエイトでバイト長であることを示していま

す。

```
DB      0B0H
DB      DAR
```

はオブジェクトコードが0B0H，PARとなることを示し，

```
ENDM
```

でコードマクロ定義を終わります。

いまの例では引数として－256～255のイミディエイトデータを受けつけます。0～99の数値は受けつけるが，それ以外はエラーとなるように作ることもできます。

```
CODEMACRO     MOVAL2   PAR:   DB(0, 99)
          DB      0B0H
          DB      DATA
ENDM
```

(0, 99) は**幅指定**といい，引数PARが0～99の間でなければならないことを指定します。このように定義した場合，

```
MOVAL2      50
```

はエラーになりませんが

```
MOVAL2      100
```

は

```
OPERAND(S)   MISMATCH   INSTRUCTION
```

のエラーとなります。

定数の引数の例として，定数をレジスタAXにかける命令MULCを作ってみましょう。8086には乗算命令がありますが，定数をかけることができません。ここでは，それに相当する命令を作ってみましょう。

```
CODEMACRO   MULC   PAR : DW
        DB     53H     ;   PUSH  BX
        DB     0BBH    ; } MOV
        DW     PAR     ; } BX, PAR
        DW     0E3F7H  ;   MUL  BX
        DB     5BH     ;   POP  BX
ENDM
```

BX レジスタを使って MULC を

```
PUSH BX
MOV BX, PAR
MUL BX
POP BX
```

でシミュレートしたわけです。この例では引数がワード長なので,

```
CODEMACRO MULC PAR : DW
```

のように引数がワード長であることを指定しています。引数は PAR を使っています。コードマクロ定義中では,これまでオブジェクトコードを **DB擬似命令**で置いてきましたが,この例のように

```
DW PAR
DW 0E3F7H
```

といった **DW擬似命令**も使えます。

レジスタが引数の場合

イミディエイトデータを引数にもつ例を示しましたが,次はレジスタを引数にもつ例をあげてみましょう。コードマクロ機能を使えば8086のすべての命令を作ることができますから,レジスタも当然引数となりえます。

そこで,

```
MOV AL, DL
```

のように

```
MOV AL
```

に相当する MOVAL2というオペコードを作ってみましょう。

レジスタを第2オペランドにもつ MOV のオブジェクトコードは少々複雑です。

```
MOV   AL, AL     88C0H
MOV   AL, AH     88E0H
MOV   AL, BL     88D8H
MOV   AL, BH     88F8H
MOV   AL, CL     88C8H
MOV   AL, CH     88E8H
```

```
MOV   AL, DL      88D0H
MOV   AL, DH      88F0H
```

第1バイトは88Hで共通ですが，第2バイトが引数のレジスタによって変わっています。オペコード表を見れば分かりますが，

AL＝0，CL＝1，DL＝2，BL＝3，

AH＝4，CH＝5，DH＝6，BH＝7

とすれば

MOV　AL

の第2バイトは

```
 7 6  5 4 3  2 1 0
|1 1  R E G |0 0 0|
```

となります。このような変則的なデータもオブジェクトコードとして生成できるように **DBIT擬似命令**が用意されています。

したがって，与えられた問題の答えは

```
CODEMACRO   MOVAL2   PAR
                    : RB────③
        DB     88H
        DBIT   2(11B),
    3(PAR(0)), 3(000B)────④
ENDM
```

③の RB は引数が**バイト長の汎用レジスタ**であることを示します。④の部分が重要です。これは，MSB から 2 ビットが11B，次の 3 ビットが PAR，残りの 3 ビットが000 B であることを示します。PAR の後ろの(0)は PAR の右シフトの必要がないので 0 となっています。

DBIT 擬似命令を使えば，ビット単位の操作ができるようになります。この例では汎用レジスタはすべて受けつけるようにしましたが，先ほどの幅指定を使えば引数のレジスタを制限できます。CL レジスタのみ引数に許すとすれば

```
CODEMACRO MOVAL2
                  PAR : RB(CL)
```

とします。こうすれば

MOVAL2　CL

は通りますが，

　MOVAL2　AL

などはまた先ほどのエラーとなってしまいます。

レジスタ名による幅指定は

　SHR　AX，CL

や

　IN　AL，DX

などCLやDXレジスタ以外はこないようなオペコードを生成するのに使えます。

メモリが引数の場合

次はメモリが引数にくる場合を考えます。

ALレジスタにメモリをロードする命令MOVAL3を作ってみましょう。

この命令を作るにはMOVのオブジェクトコードがもっとくわしく分かっている必要があります。MOVのオブジェクトコードは次のようになっています。

7 6	5 4 3	2 1 0	
1 0	0 0 1	0 D W	第1バイト
MOD	REG	R/M	第2バイト

　　　⋮

MODはアドレシングモードを示し，

　00：メモリアドレシング

　01：1バイトのディスプレースメントつきメモリアドレシング

　10：2バイトのディスプレースメントつきメモリアドレシング

　11：レジスタアドレシング

となります。

REGは使用するレジスタの種類を表し，前述のようになっています。R/Mもアドレシングモードを示し，MODとの関係で表3-2のようになります。

さらに，この2バイトの後ろに1バイトないし2バイトのディスプレースメントやオフセットがくることがあります。これらをすべてプログラマーが指定するとすればコードマクロはとても複雑なものとなってしまいますが，これら複雑な作業をすべて行ってくれる擬似命令に **MODRM擬似命令** があります。

表3-2

r/m	mod-00	mod-01	mod-10	mod-11	
				w=0	w=1
000	BX+SI	BX+SI+DISP	BX+SI+DISP	AL	AX
001	BX+DI	BX+DI+DISP	BX+DI+DISP	CL	CX
010	BP+SI	BP+SI+DISP	BP+SI+DISP	DL	DX
011	BP+DI	BP+DI+DISP	BP+DI+DISP	BL	BX
100	SI	SI +DISP	SI +DISP	AH	SP
101	DI	DI +DISP	DI +DISP	CH	BP
110	ダイレクトアドレス	BP+DISP	BP+DISP	DH	SI
111	BX	BX+DISP	BX+DISP	BH	DI

さて，問題の答えです。

```
CODEMACRO  MOVAL3
           PAR : MB———⑤
       SEGFIX  PAR
        DB        08AH
        MODRM 0, PAR
ENDM
```

⑤の MB は引数が**メモリでバイト長**であることを示します。

SEGFIX　PAR

は，この位置に PAR の存在するセグメントのオーバーライドプレフィックスを必要ならば入れることを指定しています。**SEGFIX擬似命令**を使えば，必要時に CS:，SS:，ES: などオブジェクトに生成してくれるのです。

MODRM　0，PAR

はレジスタとして 0 つまり AL レジスタを使い，PAR に合わせて MOD フィールドと R/M フィールドと必要ならばオフセットやディスプレースメントを生成せよという意味です。

引数をレジスタとメモリで別々のコードマクロを定義しましたが，これらを 1 つにすることもできます。

```
CODEMACRO  MOVAL4 PAR:EB
       SEGFIX  PAR
```

```
            DB        8AH
            MODRM   0, PAR
    ENDM
```

引数をEB，つまり実効アドレスでバイト長に宣言するだけでよいのです。

また，イミディエイトデータ，レジスタ，メモリのすべてを引数にすることもできます。

```
    CODEMACRO   MOVAL5   PAR : DB
            DB          0B0H
            DB          PAR
    ENDM
    CODEMACRO   MOVAL5 PAR : EB
            SEGFIX   PAR
            DB          08AH
            MODRM   0, PAR
    ENDM
```

となります。単に同じ名前で二重に定義しているだけですが，これでうまくいきます。

以上，コードマクロ機能の簡単な使い方を説明してみました。コードマクロ機能を使えば8086，8087の全命令を作り出すことができます。というよりも8086，8087の全命令を生成できるように作られたものがコードマクロ機能だといったほうが当たっているかもしれません。コードマクロ擬似命令の機能を見てみると，8086の命令を作るのに好都合にできています。そのため，8086を知らない人がいきなりコードマクロ擬似命令を読んでも「なんでこんな擬似命令があるのか」さっぱり分からないと思います。

また，使いこなすには，本文中でも出てきたように8086のオブジェクトコードのフォーマットが分かっている必要があります。コードマクロ機能は8086，8087の命令を使うにはとても都合がよいが，それ以外の用途にはあまり向いていないといえそうです。

コードマクロ機能はおもしろい機能ですが，8086のアセンブラを使う人がすべて知っている必要があるものだとは思いません。ただ，コードマクロを自由自在に使う人は間違いなく8086マニアだと思います。

次に，アセンブリ言語の例を示します。これは正方形の回転です。BASICによるリスト3-8と，それをアセンブリ言語にしたリスト3-9を比較してみてください。

リスト3-9はファイル名OHPC4．A86でセーブし，アセンブルから実行までの手順は次のとおりです。

ASM86　OHPC 4　⏎

GENCMD　OHPC 4　8080　⏎

OHPC 4　⏎

これからはコードマクロ擬似命令をもっとくわしく見ていきましょう。

リスト3-8　BASICによる例

```
1000 SCREEN 3,0
1010 GOSUB *SET.DATA
1020 WHILE -1
1030 FOR I=0 TO PI/2 STEP PI/30
1040   GOSUB *POLYGON
1050   IF INKEY$<>"" THEN END
1060  NEXT I
1070 WEND
1080 '
1090 *SET.DATA
1100 PI=3.141592654# : M=4 : N=8
1110 J=0 : FOR I=0 TO M-1 : READ X0(J),Y0(J) : J=J+1 : NEXT I
1120 FOR I=0 TO N-1 : READ CO(I) : NEXT I
1130 RETURN
1140 '
1150 *POLYGON
1160 GOSUB *ROT
1170 GOSUB *XDRAW.
1180  GOSUB *DRAW.
1190 RETURN
1200 '
1210 *ROT
1220  COS.=COS(I) : SIN.=SIN(I)
1230  FOR J=0 TO M-1
1240   X=X0(J) : Y=Y0(J)
1250   XD(J)=COS.*X-SIN.*Y+320
1260   YD(J)=SIN.*X+COS.*Y+200
1270 NEXT J
1280 RETURN
1290 '
1300 *DRAW.
1310 FOR J=0 TO M-1:XD1(J)=XD(J):YD1(J)=YD(J):NEXT J
1320 FOR J=0 TO N-1 STEP 2
1330   C1=CO(J) : C2=CO(J+1)
1340   X1=XD(C1) : Y1=YD(C1)
1350   X2=XD(C2) : Y2=YD(C2)
1360   LINE(X1,Y1)-(X2,Y2),7
1370 NEXT J
1380 RETURN
1390 '
1400 *XDRAW.
1410 FOR J=0 TO N-1 STEP 2
1420 C1=CO(J) : C2=CO(J+1)
```

```
1430  X1=XD1(C1) : Y1=YD1(C1)
1440   X2=XD1(C2) : Y2=YD1(C2)
1450   LINE(X1,Y1)-(X2,Y2),0
1460 NEXT J
1470 RETURN
1480 '
1490 ' DATA
1500 ' CO-ORDINATES DATA
1510 DATA 100,-100,100,100,-100,100,-100,-100
1520 ' CONNECTION DATA
1530 DATA 0,1,1,2,2,3,3,0
```

リスト3-9

```
        CSEG
        ORG     100H

; 1000 SCREEN 3,0
        MOV     AH,40H          } SCREEN  0
        INT     18H

        MOV     AH,42H          } SCREEN  3
        MOV     CH,0C0H
        INT     18H

; 1010 GOSUB *SET.DATA

; 1020 WHILE -1
WHILE:

; 1030 FOR I=0 TO PI/2 STEP PI/30
        MOV     I,0
FOR1:
        CMP     I,PI/2
        JG      NEXT1

; 1040   GOSUB *POLYGON
        CALL    POLYGON———————— 四角形描画

; 1050   IF INKEY$<>"" THEN END
        MOV     AH,01H          } キー入力チェック
        INT     18H
        OR      BH,BH
        JE      L1060

        XOR     CL,CL           } CP/M-86へ
        XOR     DL,DL
        INT     224
L1060:
; 1060   NEXT I
        ADD     I,PI/30
        JMP     FOR1
NEXT1:

; 1070   WEND
        JMP     WHILE

; 1080 '
```

```
; 1090 *SET.DATA

; 1100 PI=3.141592654# : M=4 : N=8
PI       EQU     180
M        EQU     4
N        EQU     8

; 1110 J=0 : FOR I=0 TO M-1 : READ X0(J),Y0(J) : J=J+1 : NEXT I
; 1120 FOR I=0 TO N-1 : READ CO(I) : NEXT I
; 1130 RETURN
; 1140 '
; 1150 *POLYGON
POLYGON:

; 1160 GOSUB *ROT                          } 回転変換
        CALL    ROT

; 1170 GOSUB *XDRAW.                       } 消去
        CALL    XDRAW

; 1180  GOSUB *DRAW.                       } 描画
        CALL    DRAW
; 1190 RETURN
        RET  ──────────────────────────────以下，回転のルーチン

; 1200 '
; 1210 *ROT
ROT:

; 1220  COS.=COS(I) : SIN.=SIN(I)
        MOV     AX,I
        CALL    COS_FUNC
        MOV     COS,AX

        MOV     AX,I
        CALL    SIN_FUNC
        MOV     SIN,AX

; 1230  FOR J=0 TO M-1
        MOV     J,0
FOR2:
        CMP     J,M-1
        JG      NEXT2

; 1240   X=X0(J) : Y=Y0(J)
        MOV     SI,J
        SHL     SI,1

        MOV     AX,OFFSET X0[SI]
        MOV     X,AX

        MOV     AX,OFFSET Y0[SI]
        MOV     Y,AX

; 1250   XD(J)=COS.*X-SIN.*Y+320
        MOV     CX,100

        MOV     AX,COS
        IMUL    X
        IDIV    CX
        MOV     BX,AX

        MOV     AX,SIN
        IMUL    Y
        IDIV    CX
```

```
        SUB      BX,AX
        ADD      BX,320
        MOV      OFFSET XD[SI],BX

; 1260   YD(J)=SIN.*X+COS.*Y+200
        MOV      AX,SIN
        IMUL     X
        IDIV     CX


        MOV      BX,AX

        MOV      AX,COS
        IMUL     Y
        IDIV     CX
        ADD      BX,AX
        ADD      BX,200
        MOV      OFFSET YD[SI],BX

; 1270 NEXT J
        INC      J
        JMP      FOR2
NEXT2:
; 1280 RETURN
        RET

; 1290 '
; 1300 *DRAW.
DRAW:――――――――――――――――――――――――――――――――以下，四角形描画ルーチン

; 1310 FOR J=0 TO M-1:XD1(J)=XD(J):YD1(J)=YD(J):NEXT J
        MOV      J,0
FOR3:
        CMP      J,M-1
        JG       NEXT3

        MOV      SI,J
        SHL      SI,1

        MOV      AX,OFFSET XD[SI]
        MOV      OFFSET XD1[SI],AX

        MOV      AX,OFFSET YD[SI]
        MOV      OFFSET YD1[SI],AX

        INC      J
        JMP      FOR3
NEXT3:

; 1320 FOR J=0 TO N-1 STEP 2
        MOV      J,0
FOR4:
        CMP      J,N-1
        JG       NEXT4

; 1330   C1=CO(J) : C2=CO(J+1)
        MOV      SI,J
        SHL      SI,1

        MOV      AX,OFFSET CO[SI]
        MOV      C1,AX

        MOV      AX,OFFSET CO+2[SI]
        MOV      C2,AX
```

```
; 1340   X1=XD(C1) : Y1=YD(C1)
        MOV     SI,C1
        SHL     SI,1
        MOV     AX,OFFSET XD[SI]
        MOV     X1,AX

        MOV     AX,OFFSET YD[SI]
        MOV     Y1,AX

; 1350   X2=XD(C2) : Y2=YD(C2)
        MOV     SI,C2
        SHL     SI,1
        MOV     AX,OFFSET XD[SI]
        MOV     X2,AX

        MOV     AX,OFFSET YD[SI]
        MOV     Y2,AX

; 1360   LINE(X1,Y1)-(X2,Y2),7
        MOV     BL,7
        CALL    LINE

; 1370 NEXT J
        ADD     J,2
        JMP     FOR4
NEXT4:

; 1380 RETURN
        RET

; 1390 '
; 1400 *XDRAW.
XDRAW: ──────────────────────────── 以下，四角形の消去ルーチン
; 1410 FOR J=0 TO N-1 STEP 2
        MOV     J,0
FOR5:
        CMP     J,N-1
        JG      NEXT5

; 1420 C1=CO(J) : C2=CO(J+1)
        MOV     SI,J
        SHL     SI,1

        MOV     AX,OFFSET CO[SI]
        MOV     C1,AX

        MOV     AX,OFFSET CO+2[SI]
        MOV     C2,AX

; 1430   X1=XD1(C1) : Y1=YD1(C1)
        MOV     SI,C1
        SHL     SI,1

        MOV     AX,OFFSET XD1[SI]
        MOV     X1,AX

        MOV     AX,OFFSET YD1[SI]
        MOV     Y1,AX

; 1440   X2=XD1(C2) : Y2=YD1(C2)
        MOV     SI,C2
        SHL     SI,1

        MOV     AX,OFFSET XD1[SI]
        MOV     X2,AX
```

```
        MOV     AX,OFFSET YD1[SI]
        MOV     Y2,AX

; 1450   LINE(X1,Y1)-(X2,Y2),0
        MOV     BL,0
        CALL    LINE

; 1460 NEXT J
        ADD     J,2
        JMP     FOR5
NEXT5:
; 1470 RETURN
        RET

LINE:
        PUSH    DS
        MOV     BP,X1
        MOV     CX,Y1
        MOV     DX,X2
        MOV     SI,Y2

        MOV     AX,60H
        MOV     DS,AX
        MOV     ES,AX

        MOV     .640H,BL
        MOV     .648H,BP
        MOV     .64AH,CX
        MOV     .656H,DX
        MOV     .658H,SI
        MOV     AX,0FFFFH
        MOV     .660H,AX
        MOV     AL,1
        MOV     .668H,AL
        MOV     CH,0B0H
        MOV     BX,640H
        MOV     AH,47H
        INT     18H
        POP     DS
        RET

SIN_FUNC:
;
; SIN ROUTINE
;
        CMP     AX,8000H
        JNE     SIN0
        MOV     AX,7FFFH
SIN0:
        CMP     AX,0
        JL      SINM

        CWD
        MOV     BX,360
        IDIV    BX
        CMP     DX,90
        JG      SIN1

        SHL     DX,1
        MOV     BX,DX

        MOV     AX,OFFSET SIN_TABLE[BX]
        RET

SIN1:
```

ラインルーチン
(ROM BIOSコールで直線を引く)

SINのルーチン

```
        CMP     DX,180
        JG      SIN2


        NEG     DX
        ADD     DX,180
        SHL     DX,1
        MOV     BX,DX
        MOV     AX,OFFSET SIN_TABLE[BX]
        RET

SIN2:   CMP     DX,270
        JG      SIN3
        SUB     DX,180
        SHL     DX,1
        MOV     BX,DX
        MOV     AX,OFFSET SIN_TABLE[BX]
        NEG     AX
        RET

SIN3:   NEG     DX
        ADD     DX,360
        SHL     DX,1
        MOV     BX,DX
        MOV     AX,OFFSET SIN_TABLE[BX]
        NEG     AX
        RET

SINM:   NEG     AX
        CALL    SIN
        NEG     AX
        RET

COS_FUNC:―――――――――――――――――――――――――COSのルーチン
        SUB     AX,90
        NEG     AX
        CALL    SIN_FUNC
        RET

END_CS:
        DSEG
        ORG     OFFSET END_CS
; 1480 '
; 1490 ' DATA
; 1500 ' CO-ORDINATES DATA
; 1510 DATA 100,-100,100,100,-100,100,-100,-100
X0      DW      100,100,-100,-100
Y0      DW      -100,100,100,-100

; 1520 ' CONNECTION DATA
; 1530 DATA 0,1,1,2,2,3,3,0
CO      DW      0,1,1,2,2,3,3,0
SIN_TABLE       DW      0,1,3,5,6,8,10,12,13,15
                DW      17,19,20,22,24,25,27,29,30,32
                DW      34,35,37,39,40,42,43,45,46,48
                DW      49,51,52,54,55,58,58,60,61,62
                DW      64,65,66,68,69,70,71,73,74,75
                DW      76,77,78,79,80,81,82,83,84,85
                DW      86,87,88,89,89,90,91,92,92,93
                DW      93,94,95,95,96,96,97,97,97,98
                DW      98,98,99,99,99,99,99,99,99,99
                DW      100

XD      RW      4
YD      RW      4
```

```
XD0     RW      4
YD0     RW      4
XD1     RW      4
YD1     RW      4
I       DW      0
J       DW      0
X       DW      0
Y       DW      0
X1      DW      0
Y1      DW      0
X2      DW      0
Y2      DW      0
C1      DW      0
C2      DW      0
COS     DW      0
SIN     DW      0

        END
```

12 コードマクロ擬似命令

SEGFIX擬似命令

SEGFIX 擬似命令は，例の**セグメントオーバーライドプレフィックス**を必要に応じて挿入することを指定します。セグメントオーバーライドプレフィックスは CS:，DS:，ES:，SS: のことです。本体的な例として AL レジスタにメモリの内容をロードする新しいオペコード LOAD を考えます。

```
    CODEMACRO   LOAD   X : MB
                DB          08AH
                MODRM   0, X
    ENDM
```

でよさそうですが，

```
    LOAD   VAR
    LOAD   CS:VAR
```

と書いても生成されるコードに違いが表れません。VAR のアドレスが1234H だとすれば

```
    LOAD   VAR
```

は

```
    8AH, 06H, 34H, 12H
```

というコード，

```
    LOAD   CS:VAR
```

は

```
    2EH, 8AH, 06H, 34H, 12H
```

というコードを生成するべきですが，

```
    CS  :  VAR
```

と CS: を指定してもセグメントオーバーライドプレフィックス2EH は生成されません。これを生成するのが **SEGFIX擬似命令**です。
先の例では

```
    CODEMACRO   LOAD   X : MB
```

```
        SEGFIX  X
        DB      08AH
        MODRM 0, X
    ENDM
```
とすれば，うまく働きます。

SEGFIX　形式名

には注意が必要です。形式名はメモリアドレスですから，指定子E，M，Xのいずれかで宣言されるということです。先の例では，

```
    CODEMACRO  LOAD  X : MB
```
とMで指定しています（Bはバイトの意)。

NOSEGFIX擬似命令

NOSEGFIX 擬似命令はセグメントのチェックを行います。たとえば，エクストラセグメントに属するデータしか引数にとりたくないときなどに使えます。先ほどの例で，引数をエクストラセグメントのデータしか許さないようにするには

```
    CODEMACRO  LOAD  X : MB
        NOSEGFIX  ES, X
        DB        08AH
        MODRM      0, X
    ENDM
```
とします（ただし，この場合プレフィックスはつきません)。

このように LOAD が定義された場合，

```
    LOAD  ES : VAR
```
は受けつけられますが，

```
    LOAD  VAR
    LOAD SS : VAR
```
など ES: 以外のコードは受けつけられず，

```
    ERROR  NO : 7  OPERAND(S)
        MISMATCH  INSTRUCTION
```
のエラーとなってしまいます。このように NOSEGFIX 擬似命令は生成される

コードには何の影響も与えません。単に，パラメータの正当性をチェックするためだけに使います。

NOSEGFIX　segreg,　形式名

の形式で使いますが，要は segreg と引数が一致するかどうかを調べるための擬似命令だといえます。

NOSEGFIX はストリング命令の定義に使うと考えてよいと思います。8086のストリング命令ではデスティネーションが ES: に固定されているため，デスティネーションに ES: 以外が使われないようにチェックする必要があります。NOSEGFIX を使って定義しなければならないストリング命令は

CMPS, MOVS, SCAS, STOS

ですが，そのうち MOVS のコードマクロ定義は次のようになっています。

```
CODEMACRO MOVS A:EW, B:EW
          NOSEGFIX  ES, B
          SEGFIX    A
          DB        0A5H
ENDM
```

実際問題として NOSEGFIX 擬似命令を使う機会はまずないと思います。

MODRM擬似命令

8086はレジスタやメモリをオペランドでとるときに，**MODRM バイト**がオブジェクトコードに表れます。オペコード表を見れば分かるように，これをすべてユーザーが定義していくのは大変です。これを自動的に行ってくれるのが**MODRM 擬似命令**です。

MODRM バイトは

MOD	REG	R/M

のように分けられますが，オペランドがレジスタの場合やメモリの場合，アドレシングモードによってこれらが変化します。それを勝手にやってくれるのですから，かなりユーザーは楽になります。

MODRM 擬似命令の形式は

Ⓐ MODRM　形式名，形式名

Ⓑ MODRM　NUMBER7，形式名

(NUMBER7は 0 ～ 7)

です。Ⓑは使用するレジスタを限定するときに使います。Ⓐはより一般的な形式です。例を示しましょう。

まずⒶの例です。8086のオペコード OR を作成します。

```
CODEMACRO   OR   A : RW, B : EW
            SEGFIX   B
            DB          0BH
            MODRM   A, B
ENDM
```

次にⒷの例。AL レジスタにメモリからロードする LOAD を作成します。

```
CODEMACRO   LOAD     A : MB
            SEGFIX   A
            DB        8AH
            MODRM   0, A
ENDM
```

RELB, RELW擬似命令

8086は相対ジャンプが可能ですが，そのオペコードを生成するのにぜひ必要なのが，RELB, RELW 擬似命令です。

RELB, RELW は命令の終わりとオペランドであるラベルの間のディスプレースメントを生成します。RELB はバイド，RELW はワードの変位を返します。たとえば，LOOP 命令は

```
CODEMACRO   LOOP   X : CB
            DB        0E2H
            RELB   X
ENDM
```

となります。RELB, RELW はループ，ジャンプ命令に使われることが大半で，われわれが使うことはまずないでしょう。

DB, DW, DD擬似命令

DB, DW, DD はコードマクロ定義中に展開する数値，名前を示します。

```
CODEMACRO   NOP
            DB     90H
ENDM
```

とすれば，90H を NOP が出てくるたびに展開します。

なお，DB，DW，DD は普通の擬似命令のように

```
DB  12H, 23H, 56H
```

と連記することはできません。１つひとつ

```
DB  12H
DB  23H
DB  56H
```

と書かなければならないのです。

DB，DW は形式名，数値の両方が使えますが，DD は形式名しか引数にとれません。

DBIT擬似命令

オペコードはオペランドによって１ビットだけ変化することがよくあります。ASM86はビット操作を容易に行うことのできる DBIT 命令をもっています。

DBIT の形式は複雑ですが

DBIT　〈フィールド記述〉[，〈フィールド記述〉]

フィールド記述は

〈数値〉〈組み合わせ〉

〈数値〉(〈形式名〉[〈右シフト〉])

となっています。例で示しましょう。

命令 DEC は

0 1 0 0 1	REG

という コードですが，REG の部分がレジスタによって変化します。これをコードマクロ機能で定義すると

```
CODEMACRO   DEC   REG : RW
          DBIT  5(9), 3(REG(0))
ENDM
```

となります。

DBIT　5(9), 3(REG(0))

は上位5ビットが数値の9，下位3ビットがREGを右に0回シフトしたものであることを示します。

指定子

指定子は形式パラメータの型を宣言するのに使います。

```
CODEMACRO   MOVDL   A : DB——————①
            DB   0B2H
            DB   A
ENDM
```

の①にあるDBのDが指定子に当たります。指定子には

A, C, D, E, M, R, S, X

があり，それぞれ

A：アキュムレータ（AL/AX）
C：コード（ラベル表現）
D：イミディエイトデータ
E：実効アドレス（指定子のMとRの機能）
M：メモリアドレス
R：汎用レジスタ
S：セグメントレジスタ
X：直接メモリ

となっています。

制限子

制限子も指定子と同じようにオペランドの型を指定します。いまの例では，①のDBのBが制限子です。制限子には

B, W, D, SB

があり，それぞれ

B：バイト
W：ワード

D：ダブルワード

SB：符号つきバイト

となっています。

幅指定

幅指定もオペランドを制限します。0～99の数値だけとかCXレジスタといった指定が可能です。

たとえば

CODEMACRO　OUT　X : AW, Y : RW(DX)

では第2オペランドはDXレジスタだけ許されるわけです。同様に，0～99の数値なら(0, 99)となります。幅指定で許されるのは

(NUMBERB)

(REGISTER)

(NUMBERS, NUMBERS)

(NUMBERS, REGISTER)

(REGISTER, NUMBERS)

(REGISTER, REGISTER)

です。

コードマクロをまとめてみました。率直なところ，コードマクロ機能はおもしろいが，実際に使うことは少ないのではないかと思います。コードマクロ機能はユーザーが使うためにあるというよりはアセンブラを作成する段階で派生的に生まれたもののような気がします。コードマクロ機能で新しいオペコードが自由に作れるわけですが，この新しいオペコード自体が正しいかどうか十分に検討してから使わなければなりません。アセンブルの段階ではエラーチェックができませんし，デバッガで逆アセンブルしてもソースのコードが出てくるわけではないので，デバッグも大変になるかもしれません。

もしコードマクロ機能を使うのならば，正しいと自信があるものだけを使うべきでしょう。その場その場で作るのではなく，ライブラリ化しておくほうがよいと思います。

例によってアセンブリ言語の例を示します。GDC を使って直線をランダムに描画する例です。

リスト3–10にBASICプログラム，リスト3–11にアセンブリ言語のソースを示します。

リスト3-10

```
1000 SCREEN 3,0
1010 OX=320 : OY=200
1020 FOR I=1 TO 1000
1030   GOSUB *SET.X.Y.COLOR
1040   GOSUB *DRAW.
1050 NEXT I
1060 END
2000 '
2010 *SET.X.Y.COLOR
2020  X=640*RND(1)
2030  Y=400*RND(1)
2035  COLOR.=RND(1)*6+1
2040 RETURN
3000 '
3010 *DRAW.
3020  LINE(OX,OY)-(X,Y),COLOR.
3030  OX=X : OY=Y
3040 RETURN
```

リスト3-11

```
        CSEG
        ORG     100H
; 1000 SCREEN 3,0
        MOV     AH,40H          ┐
        INT     18H             │
                                │ SCREEN3, 0をグラフィックBIOS
        MOV     AH,42H          │ をコールして実行する
        MOV     CH,0C0H         │
        INT     18H             ┘

; 1010 OX=320 : OY=200
        MOV     OX,320
        MOV     OY,200

; 1020 FOR I=1 TO 1000
        MOV     I,1
FOR:
        CMP     I,1000
        JG      NEXT

; 1030      GOSUB *SET.X.Y.COLOR
        CALL    SET_X_Y_COLOR ──────── 乱数をX, Y, COLORにセット

; 1040 GOSUB *DRAW.
        CALL    DRAW

; 1050 NEXT I
```

```
        INC     I
        JMP     FOR
NEXT:

; 1060 END
        XOR     CL,CL
        XOR     DL,DL                   } CP M-86へ
        INT     224

; 2000 '
; 2010 *SET.X.Y.COLOR
SET_X_Y_COLOR:

; 2020 X=640*RND(1)
        MOV     AX,640
        CALL    RND                     } 0~639の乱数をXにセット
        MOV     X,AX

; 2030 Y=400*RND(1)
        MOV     AX,400
        CALL    RND                     } 0~399の乱数をYにセット
        MOV     Y,AX

; 2035 COLOR.=RND(1)*6+1
        MOV     AX,6
        CALL    RND
        INC     AX                      } 1~7の乱数をCOLORにセット
        MOV     COLOR.AX

; 2040 RETURN
        RET

; 3000 '
; 3010 *DRAW.
DRAW:

; 3020 LINE(OX,OY)-(X,Y),COLOR.
        MOV     AX,OX
        MOV     X1,AX

        MOV     AX,OY
        MOV     Y1,AX
                                        } 座標のセット
        MOV     AX,X
        MOV     X2,AX

        MOV     AX,Y
        MOV     Y2,AX

; 3030  OX=X : OY=Y
        MOV     AX,X
        MOV     OX,AX

        MOV     AX,Y
        MOV     OY,AX

        CALL    LINE
; 3040 RETURN
        RET

LINE:
STATPT  EQU     0A0H ─────────────────以下，GDCを直接操作したラインルーチン
COUTPT  EQU     0A2H
POUTPT  EQU     0A0H
```

```
        MOV     COLOR0,0
        TEST    COLOR,1
        JZ      L1
        MOV     DRAWFLG,1
        JMPS    L2
L1:     MOV     DRAWFLG,0
L2:
        CALL    LINE0

        MOV     COLOR0,1
        TEST    COLOR,2
        JZ      L3
        MOV     DRAWFLG,1
        JMPS    L4
L3:     MOV     DRAWFLG,0
L4:
        CALL    LINE0

        MOV     COLOR0,2
        TEST    COLOR,4
        JZ      L5
        MOV     DRAWFLG,1
        JMPS    L6
L5:     MOV     DRAWFLG,0
L6:
        CALL    LINE0
        RET
LINE0:
TEXTW:  MOV     AL,78H
        MOV     COMMAND,AL
        CALL    OUTC
        MOV     AL,0FFH
        MOV     PARMTR,AL
        CALL    OUTP
        CALL    OUTP

;
WRITE:
        MOV     AL,DRAWFLG
        CMP     AL,1
        JNE     WRITE1

        MOV     AL,20H
        JMP     WRITE2

WRITE1:MOV      AL,22H

WRITE2:
        MOV     COMMAND,AL
        CALL    OUTC
INIT:
        MOV     AX,X2
        CMP     AX,X1
        JGE     INIT1

        MOV     BX,X1
        MOV     X1,AX
        MOV     X2,BX
        MOV     AX,Y1
        MOV     BX,Y2
        MOV     Y2,AX
        MOV     Y1,BX
INIT1:  MOV     AX,X1
        CMP     AX,X2
        JNE     INIT2
```

```
        MOV     AX,Y2
        CMP     AX,Y1
        JGE     INIT3
        MOV     BX,Y1
        MOV     Y1,AX
        MOV     Y2,BX
INIT2:
INIT3:

CULADR:
        MOV     AL,COLOR0
        CMP     AL,0
        JNE     CULI1

        MOV     BX,4000H
        JMP     CULI2

CULI1:  CMP     AL,1
        JNE     CULI3

        MOV     BX,8000H
        JMP     CULI2

CULI3:  MOV     BX,0C000H

CULI2:  MOV     EAD,BX
        MOV     AX,X1
        MOV     CL,4
        SHR     AX,CL

        ADD     AX,EAD
        MOV     EAD,AX

        MOV     AX,40
        MOV     BX,Y1
        MUL     BX

        ADD     AX,EAD
        MOV     EAD,AX

        MOV     AX,X1
        AND     AX,0FH
        MOV     DAD,AL

;
CSRW:
        MOV     AL,49H
        MOV     COMMAND,AL
        CALL    OUTC

        MOV     AX,EAD
        MOV     PARMTR,AL
        CALL    OUTP
        MOV     PARMTR,AH
        CALL    OUTP

        MOV     AL,DAD
        MOV     CL,4
        SHL     AL,CL
        MOV     PARMTR,AL
        CALL    OUTP
;
CULDIR:
        MOV     AX,X2
        SUB     AX,X1
        MOV     DELTA_X,AX
```

```
        MOV     AX,Y2
        SUB     AX,Y1
        MOV     DELTA_Y,AX

        MOV     AX,DELTA_Y
        CMP     AX,0
        JLE     DIR1

        MOV     BX,DELTA_X
        CMP     AX,BX
        JLE     DIR2

        XOR     AL,AL
        MOV     DIR,AL

        MOV     AX,DELTA_X
        MOV     BX,DELTA_Y
        MOV     DELTA_Y,AX
        MOV     DELTA_X,BX

        JMP     DIR3

DIR2:   MOV     AL,1
        MOV     DIR,AL
        JMP     DIR3
DIR1:   MOV     BX,DELTA_Y
        NEG     BX
        MOV     DELTA_Y,BX

        MOV     AX,DELTA_X
        CMP     BX,AX
        JG      DIR4

        MOV     AL,2
        MOV     DIR,AL
        JMP     DIR3

DIR4:   MOV     AL,3
        MOV     DIR,AL

        MOV     AX,DELTA_X
        MOV     BX,DELTA_Y
        MOV     DELTA_Y,AX
        MOV     DELTA_X,BX

DIR3:
;
CULPAR:
        MOV     AX,DELTA_X
        MOV     DC_REG,AX

        MOV     AX,DELTA_Y
        SAL     AX,1
        SAL     AX,1
        SUB     AX,DELTA_X
        MOV     D_REG,AX

        MOV     AX,DELTA_Y
        SUB     AX,DELTA_X
        SAL     AX,1
        SAL     AX,1
        MOV     D2_REG,AX

        MOV     AX,DELTA_Y
        SAL     AX,1
        MOV     D1_REG,AX
;
VECTW:
        MOV     AL,4CH
        MOV     COMMAND,AL
        CALL    OUTC

        MOV     AL,DIR
        OR      AL,8
        MOV     PARMTR,AL
        CALL    OUTP

        MOV     AX,DC_REG
        MOV     WRDPAR,AX
        CALL    OUTWP
        MOV     AX,D_REG
        MOV     WRDPAR,AX
        CALL    OUTWP

        MOV     AX,D2_REG
        MOV     WRDPAR,AX
        CALL    OUTWP

        MOV     AX,D1_REG
        MOV     WRDPAR,AX
        CALL    OUTWP
;
VECTE:
        MOV     AL,6CH
        MOV     COMMAND,AL
        CALL    OUTC
;
        RET
;
;       END OF LINE ROUTINE
;

;
; COMMAND OUT ROUTINE FOR GDC
;
OUTC:
;
OUTCLP: IN      AL,STATPT
        AND     AL,2
        JNZ     OUTCLP

        MOV     AL,COMMAND
        OUT     COUTPT,AL
        RET

;
; PARAMETER OUT ROUTINE FOR GDC
;
OUTP:

OUTPLP: IN      AL,STATPT
        AND     AL,2
        JNZ     OUTPLP

        MOV     AL,PARMTR
        OUT     POUTPT,AL
        RET

OUTWP:
        MOV     AX,WRDPAR
        MOV     PARMTR,AL
        CALL    OUTP
```

```
        MOV     PARMTR,AH
        CALL    OUTP

        RET
RND:
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     CX,AX
        MOV     AX,259
        MUL     SEED
        ADD     AX,3
        AND     AX,32767
        MOV     SEED,AX
        MUL     CX
        MOV     BX,32767
        DIV     BX

        POP     DX
        POP     CX
        POP     BX
        RET

END_CS:
        DSEG
        ORG     OFFSET END_CS

I       DW      0
COLOR   DW      0
```

```
X       DW      0
Y       DW      0

OX      DW      0
OY      DW      0

X1      DW      0
Y1      DW      0

X2      DW      0
Y2      DW      0

SEED    DW      1234

COMMAND DB      0
PARMTR  DB      0
DRAWFLG DB      0
COLOR0  DB      0
EAD     DW      0
DAD     DB      0
DELTAX  DW      0
DELTAY  DW      0
DIR     DB      0
DREG    DW      0
D1REG   DW      0
D2REG   DW      0
DCREG   DW      0
WRDPAR  DW      0

        END
```

13 ブートフロッピーの作成

ここではASM86の結びとして，ブートフロッピーの作成を行います。

私たちがASM86を使って作ったプログラムはCP/M-86の下で動作します。ASM86. CMD, GENCMD. CMDを使えば，簡単に実行可能なファイルを作ることができます。CP/M-86の上で動作するだけで十分な場合もありますが，市販用のゲームプログラムなどがCP/M-86上で作動するのでは困ります。ディスクを入れてリセットすれば自動的にプログラムがロードされ，ゲームが始まらなければなりません。ここではこのブートフロッピーを作成するうえで必要なプログラムをASM86で記述してみます。対象は5インチ倍トラックとしますが，8インチや5インチ単密度/高密度でもそのノウハウは同じです。

PC-9801のブートストラップ動作

PC-9801は電源を入れるとどうなるのでしょうか。CPU8086はリセットされると，セグメント0FFFFH，オフセット0000Hから実行を開始します。メモリマップを見るまでもなく，この部分はROMです。ですから，9801はリセット直後はROMのプログラムを実行しているはずです。

ROM内では，メモリの実装状態のチェックや割り込みベクトルの初期化，CPUのレジスタのチェックなどさまざまな初期化，診断が行われます。

9801には**BIOS**と呼ばれるサービスルーチンがあり，グラフィックス，キーボードの入出力，ディスクの入出力などがプログラムを書かずに行えます。これらBIOSもリセット時点で使えるようになります。つまり，OSを立ち上げなくてもBIOSはユーザープログラムから使うことができるのです。これは非常に便利なことです。1文字入出力ルーチンなどをユーザーが一から作る必要がないのですから。

初期化などが終わったら，メモリスイッチ5番に従ってシステムの立ち上げにかかります。5インチ2DDからの立ち上げを指定し，ディスクが倍密度256バイト/セクタでフォーマットしてある場合，

　トラック0

　サーフィス0

セクタ1, 2, 3, 4

の1024バイトのデータをIPL(イニシャルプログラムローダ)とみなし，この1024バイトをセグメント1FC0H，オフセット0000H以降にロード，セグメント1FC0H，オフセット0000Hに飛び込みます。1024バイト以内の短いプログラムならばトラック0，サーフィス0，セクタ1，2，3，4に書き込めばブートフロッピーになりますが，普通のプログラムは1024バイトを楽に超えてしまいます。ですから，普通はプログラムをディスクの別の部分に書いておき，IPLがそのプログラムをロードし，実行するようにします。さらに大きく複雑な場合はIPLがプログラムをロードし，そのプログラムがメインプログラムをロード，実行するようにします。

つぎにIPLがメインプログラムをロードして実行するブートフロッピーを実際に作成しましょう。

実際にブートフロッピーを作る

PC-9801のブートストラップが分かりました。実際にブートフロッピーを作成するには，次の4つのステップを踏むことになります。

- IPLを作る
- IPLを書き込む
- メインプログラムを作る
- メインプログラムを書き込む

メディアには5インチ倍トラックディスクを使用し，全トラックが256バイト/セクタでフォーマットされていることを前提とします。CP/M-86やN88-DISK BASICのディスクは256バイト/セクタですから，いままで使っていたメディアが流用できます。MS-DOSの5インチディスクは1024バイト/セクタですからCP/M-86でフォーマットしてから使ってください。また，Aドライブから立ち上げることにします。

メインプログラムは8080モデルで作成し，

トラック1～16

サーフィス0

に配置します。セクタの順番をとびとびにする**セクタスキュー**は行わず，素直に

1，2，3，…16

の順に書き込むことにします。また，IPLによってセグメント0800H，オフセット0000H以降にロード，実行することにしましょう。

IPLを作る

話をすべて上の条件に限定して進めます。

IPLは

サーフィス0

トラック1～16

にあるメインプログラムを物理アドレスのセグメント0800H，オフセット0000H以降にロードしなければなりません。問題なのはサーフィス0，トラック1

〜16にあるデータをどのようにして読むかです。BASIC なら DSKI$ で簡単に行えますが，機械語ではそう簡単にはいきません。

CP/M-86や MS-DOS では BIOS をコールしたりアブソリュートディスクリードをすればよいのですが，「IPL 動作時は何の OS も動作していない」ため，いかなる OS の BIOS やサービスルーチンも使用できません。最悪の場合，**FDC**（フロッピーディスクコントローラ），**PIC**（割り込みコントローラ），**DMAC**(DMA コントローラ）にコマンドを送り，データを受け取るプログラムを一から作成しなければなりません。ところが，9801は IPL 動作時にも ROM BIOS が生きています。

先にも述べたとおり，この BIOS の中にはディスクの入出力も含まれていますから，DIKI$ の機能が機械語で比較的簡単に行えます。

BIOS コマンドの「データの読み出し」の方法は，レジスタを

	7	6	5	4	3	2	1	0
AH	MT	MF	$\bar{r}$	SEEK	0	1	I	0

	7	6	5	4	3	2	1	0
AL	0	1	1	1	0	0	X_2	X_1

BX　データ長
CL　シリンダ番号
DH　ヘッド番号
DL　セクタ番号
CH　セクタ長
ES:BP　転送先

のようにセットし，

INT　1BH

を実行するだけです。レジスタにセットするデータに見慣れないものもあるので，説明しましょう。

AH のビット7にある MT は，シングルトラックで読み出すかマルチトラックで読み出すかの指定です。ここではプログラムを簡単にするため，シングルトラック読み出しとします。

ビット6のMFは読み込むデータが倍密度か単密度かを指定します。倍密度をMFM，単密度をFMなんだと思ってください。

ビット5の$\bar{r}$はエラー時のリトライを行うかどうかのフラグです。バーがついていることからも分かるとおり0でリトライし，1でリトライしない指定となります。なお，リトライは8回行われます。

ビット4のSEEKは現在ヘッドのある位置から読むか，シークして指定トラックから読むかの設定です。ここでは簡単に1とします。

以上の設定から，ここでは

AH＝0 1 0 1 0 1 1 0
　＝56H

となります。

ALのビット1，0のX_2，X_1はディスクドライブの番号です。ここではドライブAから立ち上げるので

0 0

とします。したがって，

AL＝0 1 1 1 0 0 0 0
　＝70H

となります。

BXのデータ長は一度に転送してくるデータ数で，ここでは簡単にするため256バイト（1セクタ）ずつの転送とします。よって，

BX＝256
　＝0 1 0 0 H

CLのシリンダ番号はトラック番号のことです。FDCのマニュアルではトラック番号といわず，シリンダ番号と表現しています。

DHのヘッド番号はサーフィス番号のことです。

CHのセクタ長は0～3で，それぞれ128バイト/セクタ，256バイト/セクタ，512バイト/セクタ，1024バイト/セクタに対応しています。ここでは256バイト/セクタですから1とします。したがって，

CH＝1

となります。

上記のようにレジスタを設定して

```
INT  1BH
```
を実行するとディスク上のデータがメモリに転送されますが，エラーが起きた場合，キャリーフラグCFが1となり，AHレジスタにどのようなエラーがあったかを返します。ここではエラー処理は行いません。

このBIOSの機能を利用してDSKI$に相当するサブルーチンを作れば，IPL作成は簡単になります。

```
DSKI:
        MOV  AH, 56H
        MOV  AL, 70H
        MOV  BX, 100H
        MOV  CL, TRACK
        MOV  DH, 0
        MOV  DL, SECTOR
        MOV  CH, 1
        MOV  BP, 800H
        MOV  ES, BP
        MOV  BP, ADDRESS
        INT  1BH
        RET
```
となるでしょう。

IPL本体で注意することは，セグメントレジスタの初期化が必要なことです。CP/M-86やMS-DOSの下でユーザープログラムを実行する場合，OSがセグメントレジスタの設定を行いますが，IPLが実行されるときには設定はされません。これはIPLで確実に行います。また，スタックポインタのセットもIPLで行うほうが無難でしょう。

スタックエリアをIPL1024バイトの後半に割り当てます。IPLはセグメント1FC0H，オフセット0000Hからロードされ，同じアドレスから実行されるので，

```
ORG  0000H
```
としなければなりません。CP/M-86のように，

```
ORG  100H
```

とすると暴走してしまうでしょう。

以上より，IPL はリスト3-12のようになります。

リスト3-12 ソースファイル名 IPLA86

```
;
; IPL ( INITIAL PROGRAM LOADER )
        CSEG    1FC0H
IPL:
        MOV     AX,CS
        MOV     DS,AX

        CLI
        MOV     SS,AX
        MOV     SP,0FFFFH
        STI

        MOV     ADDRESS,0
        MOV     TRACK,1
FOR1:
        CMP     TRACK,16
        JG      NEXT1
        MOV     SECTOR,1
FOR2:
        CMP     SECTOR,16
        JG      NEXT2

        CALL    DSKI
        ADD     ADDRESS,256

        INC     SECTOR
        JMPS    FOR2
NEXT2:
        INC     TRACK
        JMPS    FOR1
NEXT1:
        JMPF    MAIN

DSKI:
;
; ONE SECTOR READ
;
        MOV     AH,56H
        MOV     AL,70H
        MOV     BX,100H
        MOV     CL,TRACK
        MOV     DH,0
        MOV     DL,SECTOR
        MOV     CH,1
        MOV     BP,800H
        MOV     ES,BP
        MOV     BP,ADDRESS
        INT     1BH
        RET
END_CS:
        DSEG    1FC0H
        ORG     OFFSET END_CS
TRACK   RB      1
SECTOR  RB      1
ADDRESS RW      1

        CSEG    800H
        ORG     100H
MAIN:
        END
```

SS, SP レジスタをセットするときは，インタラプトをマスクしてください(CLI)。また，設定終了後はただちにマスクを解いてください(STI)。さらに，メインルーチンに飛び込むときに FAR ジャンプ(JMPF)を使っている点にも注意してください。メインプログラムはデバッグなどを考慮して

ORG　100H

で実行するようにしました。こうすれば，CP/M-86上で DDT86や SID86を使ったデバッグができますし，そのままブートフロッピーに書き込むことができます。

このプログラムを IPL．A86というファイル名でセーブして

ASM86 IPL　⏎

GENCMD IPL 8080　⏎

とすれば，IPL．CMD のでき上がりです。

プログラム中で

```
CSEG    1FC0H
DSEG    1FC0H
CSEG    800H
```

のようなセグメント擬似命令プラス数値式を使いました。CP/M-86上で実行するプログラムでこれを指定すると実行できないと前に述べましたが，ここではCP/M-86上で実行するわけではありませんので自由に使えるわけです。ソースプログラム中に挿入したBASIC ステートメントも参考にしてください。

IPLを書き込む

IPL は完成しましたが，今度はこれを

トラック0

セクタ1，2，3，4

に書き込まなければなりません。そのためには，

- CMD ファイルの構造
- ファイルを読む方法
- ディスクに書き込む方法

が分からなければなりません。「IPL を書き込む」作業はCP/M-86上で行いますので，CP/M-86の BDOS コール，BIOS コールや高級言語を使ってもかまいません。でも，そんなに複雑な作業ではないので，ASM86を使って行うことにします。

CMDファイルの構造

CMD ファイルの構造はMS-DOS の EXE ファイルに比べると非常に簡単です。

先頭128バイトがヘッダレコードと呼ばれ，各セグメントの種類，大きさ，メモリ領域の最大値，最小値を保持し，その残りが**メモリイメージ**となっています。メモリイメージは，そのままの型でメモリにロードすれば即実行可能ですから「メモリイメージをトラック0，セクタ1，2，3，4に書き込めばよい」わけです。

ファイルを読む方法

次に，IPL，CMD を読み込まなければなりません。BASIC ならば

OPEN "IPL. CMD" FOR INPUT AS #1

などとして INPUT$ などで簡単に読めますが，機械語では CP/M-86の BDOS コールで行うのが普通でしょう。使うのは BDOS コールの15番ファイルのオープンと20番シーケンシャルな読み出しの機能です。

ファイルオープンの方法は

```
CL          15（機能コード）
DX          FCB のオフセット
```

をセットし，

```
INT  224
```

を実行します。リターンコードは AL が 0，1，2，3 ならば正常終了，0FFH ならファイルが発見できなかったことを表します。

FCB（ファイルコントロールブロック）は CP/M-86でファイルを操作するときに必ず出てくるものです。ここではデフォルト FCB を使うので，FCB のオフセットは5BH，すなわち

DX＝5BH

とします。

「シーケンシャルな読み出し」機能はファイルをシーケンシャルに128バイトずつメモリに読み込む機能で，方法は

```
CL          20（機能コード）
DX          FCB のオフセット
```

をセットし

```
INT          224
```

とします。転送先は **DMAバッファ** と呼ばれ，デフォルト状態でオフセット80Hの位置にあります。

なお，シーケンシャルな読み出しを行う最初に，FCB 内の cr フィールドを 0 にする必要があります。これは，FCB の先頭から32バイト目に当たります。

リターンコードは AL が 0 で正常，1 でノーデータです。

ディスクに書き込む方法

トラック0，サーフィス0，セクタ1，2，3，4に書くにはROM BIOSを直接コールするのが簡単でしょう。IPL. A86中のDSKIルーチンのAHレジスタを

01010101＝55H

と変更すれば「データの書き込み」になります。

これでIPL書き込み用プログラムが作成できるはずです。これをWIPL. A86（リスト3-13）とすれば，

ASM86 WIPL ⏎

GENCMD WIPL 8080 ⏎

で実行可能なCMDファイルができます。そしてドライブAにブートフロッピー（これから書き込むディスク）を入れ，

B:WIPL B:IPL. CMD ⏎

とすればドライブAのディスクにIPLが書き込まれます。なお，このときドライブBにはWIPL. CMD，IPL. CMDが入っている必要があります。

リスト3-13

```
;
; WRITE IPL
;
        CSEG
        ORG     100H
WIPL:
        MOV     CL,15
        MOV     DX,5CH
        INT     224

        MOV     BYTE PTR .5CH+32,0

        MOV     TRACK,0
        MOV     SECTOR,1

        CALL    READ_SEQUENCIAL
L1:
        CMP     SECTOR,4
        JG      FIN

        CALL    READ_DATA

        PUSH    AX
        CALL    DSKO
        POP     AX

        CMP     AL,1
        JE      FIN
```

```
        INC     SECTOR
        JMPS    L1
FIN:
        XOR     BL,BL
        XOR     CL,CL
        INT     224

READ_DATA:
        CALL    READ_SEQUENCIAL

        MOV     DI,OFFSET BUFFER
        CALL    MOVE

        CALL    READ_SEQUENCIAL
        MOV     DI,OFFSET BUFFER+80
        CALL    MOVE
        RET

READ_SEQUENCIAL:
        MOV     CL,20
        MOV     DX,5CH
        INT     224
        RET

MOVE:
        CLD
        MOV     BX,DS
        MOV     ES,BX
        MOV     SI,80H
```

```
        MOV     CX,128
        REP     MOVSB
        RET

DSK0:
;
; ONE SECTOR WRITE
;
        MOV     AH,55H
        MOV     AL,70H
        MOV     BX,100H
        MOV     CH,1
        MOV     CL,TRACK
        MOV     DH,0
```

```
        MOV     DL,SECTOR
        MOV     BP,DS
        MOV     ES,BP
        MOV     BP,OFFSET BUFFER
        INT     1BH
        RET
END_CS:
        DSEG
        ORG     OFFSET END_CS

BUFFER  RB      256
TRACK   DB      0
SECTOR  DB      1

        END
```

メインプログラムを作る

メインプログラムは何でもいいのですが，OSのシステムコールやBIOSを使わずに書かなければなりません。唯一使っていいのはROM BIOSだけです。また，ここでのIPLはメインプログラムロード後，セグメント800H，オフセット100Hに飛び込むようにしたので，メインプログラムも

```
        CSEG    800H
        ORG     100H
MAIN:
```

としなければなりません。

そして，メインルーチンでもセグメントレジスタ，スタックポインタの初期化をきっちり行う必要があります。

メインプログラムを書き込む

IPLはサーフィス0，トラック1～16のデータをロードしますから，メインプログラムをここに書き込まなければなりません。これは，明らかにWIPL.A86とほとんど同じプログラムになります。

メインプログラムを書き込むプログラムをWMAIN.A86（リスト3-14），メインプログラムをMAIN.A86（リスト3-15）とすれば

ASM86 MAIN ⏎

GENCMD MAIN 8080 ⏎

ASM86 WMAIN ⏎

GENCMD WMAIN 8080 ⏎

とし，MAIN. CMD, WMAIN. CMD をドライブBに，ブートフロッピーをドライブAに入れ，

B:WMAIN B:MAIN. CMD ⏎

とすれば，めでたくブートフロッピーの完成です。あとは本体の電源を切り，ふたたび入れてブートフロッピーをドライブに入れれば，OS もないのに勝手に立ち上がり，勝手にメインプログラムが始まります。

リスト3-14

```
;
; WRITE MAIN PROGRAM
;
        CSEG
        ORG     100H
WMAIN:
        MOV     CL,15
        MOV     DX,5CH
        INT     224

        MOV     BYTE PTR.5CH+32,0

        CALL    READ_SEQUENCIAL

        MOV     TRACK,1
FOR1:
        CMP     TRACK,16
        JG      NEXT1

        MOV     SECTOR,1
FOR2:
        CMP     SECTOR,16
        JG      NEXT2

        CALL    READ_DATA

        PUSH    AX
        CALL    DSKO
        POP     AX

        CMP     AL,1
        JE      FIN

        INC     SECTOR
        JMPS    FOR2
NEXT2:
        INC     TRACK
        JMPS    FOR1
NEXT1:

FIN:
        XOR     BL,BL
        XOR     CL,CL
        INT     224

READ_DATA:
        CALL    READ_SEQUENCIAL
        MOV     DI,OFFSET BUFFER
        CALL    MOVE

        CALL    READ_SEQUENCIAL
        MOV     DI,OFFSET BUFFER+80H
        CALL    MOVE
        RET

READ_SEQUENCIAL:
        MOV     CL,20
        MOV     DX,5CH
        INT     224
        RET

MOVE:
        CLD
        MOV     BX,DS
        MOV     ES,BX
        MOV     SI,80H
        MOV     CX,128
        REP     MOVSB
        RET

DSKO:
;
; ONE SECTOR WRITE
;
        MOV     AH,55H
        MOV     AL,70H
        MOV     BX,100H
        MOV     CH,1
        MOV     CL,TRACK
        MOV     DH,0
        MOV     DL,SECTOR
        MOV     BP,DS
        MOV     ES,BP
        MOV     BP,OFFSET BUFFER
        INT     1BH
        RET
END_CS:
        DSEG
        ORG     OFFSET END_CS

BUFFER  RB      256
TRACK   DB      0
SECTOR  DB      1

        END
```

リスト3-15

```
;
; MAIN PROGRAM
;
; POLYGON TRANSFORMATION
;

; 1000 MAX=8
MAX      EQU      8

; 1010 VMAX=20
VMAX     EQU      20

; 1015 X1=320-150 : Y1=200-150
X1       EQU      320-150
Y1       EQU      200-150

; 1017 X2=320+150 : Y2=200+150
X2       EQU      320+150
Y2       EQU      200+150

         CSEG     800H          } メインプログラムはセグメント800H,
         ORG      100H          } オフセット100Hからロード

         MOV      AX,CS         |
         MOV      DS,AX         |
                                | セグメントスタック
         CLI                    | の初期化
         MOV      SS,AX         |
         MOV      SP,0FFFFH     |
         STI

; 1030 SCREEN 3,0
         MOV      AH,40H        |
         INT      18H           |
                                | ROM BIOSによるスクリーンモードのセット
         MOV      AH,42H        |
         MOV      CH,0C0H       |
         INT      18H

; 1040 GOSUB *SET.X.Y.VX.VY.C
         CALL     SET_X_Y_VX_VY_C

; 1050 FOR J=1 TO MAX
L1050:
         MOV      J,1
FOR1:
         CMP      J,MAX
         JLE      L1060
         JMP      NEXT1

; 1060   I=J-1 : IF J=1 THEN I=MAX
L1060:
         MOV      AX,J
         DEC      AX
         MOV      I,AX

         CMP      J,1
         JNE      L1070

         MOV      I,MAX

; 1070 K=J+1 : IF J=MAX THEN K=1
L1070:
         MOV      AX,J
```

```
        INC     AX
        MOV     K,AX

        CMP     J,MAX
        JNE     L1080

        MOV     K,1

; 1080 OX1=X(I) : OY1=Y(I)
L1080:
        MOV     SI,I
        SHL     SI,1

        MOV     AX,X[SI]
        MOV     OX1,AX

        MOV     AX,Y[SI]
        MOV     OY1,AX

; 1090  OX2=X(J) : OY2=Y(J)
        MOV     SI,J
        SHL     SI,1

        MOV     AX,X[SI]
        MOV     OX2,AX

        MOV     AX,Y[SI]
        MOV     OY2,AX

; 1100   OX3=X(K) : OY3=Y(K)
        MOV     SI,K
        SHL     SI,1

        MOV     AX,X[SI]
        MOV     OX3,AX

        MOV     AX,Y[SI]
        MOV     OY3,AX

; 1110   LINE(OX1,OY1)-(OX2,OY2),0
        MOV     AX,OX1
        MOV     XX1,AX

        MOV     AX,OY1
        MOV     YY1,AX

        MOV     AX,OX2          ラインルーチンをコール
        MOV     XX2,AX          (消去)

        MOV     AX,OY2
        MOV     YY2,AX

        MOV     COLOR,0

        CALL    LINE

; 1120   LINE(OX2,OY2)-(OX3,OY3),0
        MOV     AX,OX2
        MOV     XX1,AX

        MOV     AX,OY2
        MOV     YY1,AX

        MOV     AX,OX3
        MOV     XX2,AX          ラインルーチンをコール
                                (消去)
        MOV     AX,OY3
```

```
        MOV     YY2,AX

        MOV     COLOR,0

        CALL    LINE

; 1130  VX=VX(J): VY=VY(J)
        MOV     SI,J
        SHL     SI,1

        MOV     AX,VX[SI]
        MOV     VX,AX

        MOV     AX,VY[SI]
        MOV     VY,AX

; 1140 C1=C(I) : C2=C(J)
        MOV     SI,I
        SHL     SI,1


        MOV     AX,C[SI]
        MOV     C1,AX

        MOV     SI,J
        SHL     SI,1

        MOV     AX,C[SI]
        MOV     C2,AX

; 1150   X=OX2+VX : Y=OY2+VY
        MOV     AX,OX2
        ADD     AX,VX
        MOV     X,AX

        MOV     AX,OY2
        ADD     AX,VY
        MOV     Y,AX

; 1160  IF X<X1 THEN X=X1 : VX=-VX
        CMP     X,X1
        JGE     L1170
        MOV     X,X1
        NEG     VX
; 1170   IF X>X2 THEN X=X2 : VX=-VX
L1170:
        CMP     X,X2
        JLE     L1180

        MOV     X,X2
        NEG     VX

; 1180 IF Y<Y1 THEN Y=Y1 : VY=-VY
L1180:
        CMP     Y,Y1
        JGE     L1190

        MOV     Y,Y1
        NEG     VY

; 1190 IF Y>Y2 THEN Y=Y2 : VY=-VY
L1190:
        CMP     Y,Y2
        JLE     L1200

        MOV     Y,Y2
        NEG     VY
```

```
; 1200 LINE(OX1,OY1)-(X,Y),C1
L1200:
        MOV     AX,OX1
        MOV     XX1,AX

        MOV     AX,OY1
        MOV     YY1,AX

        MOV     AX,X
        MOV     XX2,AX

        MOV     AX,Y
        MOV     YY2,AX

        MOV     AX,C1
        MOV     COLOR,AX

        CALL    LINE
```
C1のカラーコードで直線描画

```
; 1210  LINE(X,Y)-(OX3,OY3),C2
        MOV     AX,X
        MOV     XX1,AX

        MOV     AX,Y
        MOV     YY1,AX

        MOV     AX,OX3
        MOV     XX2,AX

        MOV     AX,OY3
        MOV     YY2,AX

        MOV     AX,C2
        MOV     COLOR,AX

        CALL    LINE
```
C2のカラーコードで直線描画

```
; 1220  X(J)=X : Y(J)=Y
        MOV     SI,J
        SHL     SI,1

        MOV     AX,X
        MOV     X[SI],AX

        MOV     AX,Y
        MOV     Y[SI],AX

; 1230   VX(J)=VX : VY(J)=VY
        MOV     AX,VX
        MOV     VX[SI],AX

        MOV     AX,VY
        MOV     VY[SI],AX

; 1240 NEXT J
        INC     J
        JMP     FOR1
NEXT1:

; 1250 GOTO 1050
        JMP     L1050


; 1260
; 1270 *SET.X.Y.VX.VY.C
SET_X_Y_VX_VY_C:
```

```
; 1280 FOR I=1 TO MAX
        MOV     I,1
FOR2:
        CMP     I,MAX
        JG      NEXT2

; 1290  X(I)=INT(RND(1)*640)
L1290:
        MOV     SI,I
        SHL     SI,1

        MOV     AX,640
        CALL    RND

        MOV     X[SI],AX

; 1300  Y(I)=INT(RND(1)*400)
        MOV     AX,400
        CALL    RND

        MOV     Y[SI],AX

; 1310   VX(I)=INT(RND(1)*VMAX)-VMAX/2
        MOV     AX,VMAX
        CALL    RND
        SUB     AX,VMAX/2

        MOV     VX[SI],AX

; 1320  VY(I)=INT(RND(1)*VMAX)-VMAX/2
        MOV     AX,VMAX
        CALL    RND
        SUB     AX,VMAX/2

        MOV     VY[SI],AX

; 1330 C(I)=INT(RND(1)*7+1)
        MOV     AX,7
        CALL    RND
        INC     AX

        MOV     C[SI],AX

; 1340 NEXT I
        INC     I
        JMP     FOR2
NEXT2:
; 1340 RETURN
        RET

LINE:
        PUSH    DS
        MOV     BP,XX1
        MOV     CX,YY1
        MOV     DX,XX2
        MOV     SI,YY2
        MOV     BX,COLOR
        MOV     AX,60H
        MOV     DS,AX
        MOV     ES,AX

        MOV     .640H,BL
        MOV     .648H,BP
        MOV     .64AH,CX
        MOV     .656H,DX
        MOV     .658H,SI
```

配列X, Y, VX, VY, Cを
乱数で初期化

ラインルーチン
(ROM BIOSをコール)

```
        MOV     AX,0FFFFH
        MOV     .660H,AX
        MOV     AL,1
        MOV     .668H,AL
        MOV     CH,0B0H
        MOV     BX,640H
        MOV     AH,47H
        INT     18H
        POP     DS
        RET

RND:
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     CX,AX
        MOV     AX,259
        MUL     SEED
        ADD     AX,3
        AND     AX,32767
        MOV     SEED,AX                 乱数ルーチン
        MUL     CX
        MOV     BX,32767
        DIV     BX

        POP     DX
        POP     CX
        POP     BX
        RET

END_CS:
        DSEG    800H
        ORG     OFFSET END_CS

; 1020 DIM X(MAX9,Y(MAX),VX(MAX),VY(MAX),C(MAX)
X       RW      MAX+1
Y       RW      MAX+1
VX      RW      MAX+1                   配列
VY      RW      MAX+1
C       RW      MAX+1

OX1     DW      0
OY1     DW      0

OX2     DW      0
OY2     DW      0

OX3     DW      0
OY3     DW      0

I       DW      0
J       DW      0
K       DW      0

XX1     DW      0
YY1     DW      0
XX2     DW      0
YY2     DW      0
COLOR   DW      0

C1      DW      0
C2      DW      0

SEED    DW      1234

        END
```

CP/M-86のASM86でブートフロッピーを作る手順を述べました。ASM86でもこのようにCP/M-86とは無関係な(かつ有用な)プログラムが開発できることが分かったと思います。同様な方法で他機種のプログラムを9801で作ることもできるはずです。

ASM86以外のアセンブラ 15

ここでASM86以外のアセンブラを紹介しましょう。

MASM

MS-DOSを買うとついてくるのがMACRO86です。これはMASM. EXEというファイルになっています。MASMは**リロケータブル**で**マクロ機能**をもっています。8086のアセンブラとしては十分すぎるほどの機能といえ，これがタダでついてくるのは驚きに値します。

MACRO86の使い方

MACRO86のソースプログラムをTEMP. ASM（リスト3-16）とします。拡張子は.ASMとするのが普通です。

リスト3-16 MACRO86のソースプログラムの例

```
;
; SERCH FILE & DIRECTORY NAME
;
ERROR   EQU     00H
NORMAL  EQU     NOT     ERROR

ENTRY   MACRO
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    SI
        PUSH    DI
        PUSH    BP
        PUSH    DS
        PUSH    ES

        MOV     AX,CS
        MOV     DS,AX
        ENDM

EXIT    MACRO
        LOCAL   L1,L2
        JNC     L1
        MOV     COND,ERROR
        JMP     SHORT L2
L1:     MOV     COND,NORMAL
L2:
        POP     ES
        POP     DS
        POP     BP
        POP     DI
        POP     SI
```

```
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        IRET

        ENDM

CODE    SEGMENT PUBLIC
        ASSUME  CS:CODE,DS:CODE
        ORG     0000H
        ENTRY

        MOV     AH,1AH  ; SET DTA
        MOV     DX,OFFSET DTA
        INT     21H

        MOV     AH,4EH  ; FIND MATCH FILE
        MOV     DX,OFFSET PATH_NAME
        MOV     CX,17H
        INT     21H

NEXT:
        ENTRY

        MOV     AH,1AH  ; SET DTA
        MOV     DX,OFFSET DTA
        INT     21H

        MOV     AH,4FH  ; STEP THROUGH A DIRECTORY MATCHING FILES
        INT     21H

        EXIT

        ORG     100H
DTA     DB      128 DUP(?)
        ORG     200H
COND    DB      NORMAL
PATH_NAME       DB      '*.*',0

CODE    ENDS
        END
```

リスト3-17 TEMP. OBJの内容

```
Dump Version 2.0

00000000  80 03 00 01 41 3B 96 07-00 00 04 43 4F 44 45 44   ....A;.....CODED
00000010  98 07 00 68 05 02 02 01-01 EE A0 54 00 01 00 00   ...h...... T....
00000020  50 53 51 52 56 57 55 1E-06 8C C8 8E D8 B4 1A BA   PSQRVWU...ﾈ.ﾘｴ.ｺ
00000030  00 00 CD 21 B4 4E BA 00-00 B9 17 00 CD 21 50 53   ..ﾍ!ｴNｺ..ｹ..ﾍ!PS
00000040  51 52 56 57 55 1E 06 8C-C8 8E D8 B4 1A BA 00 00   QRVWU...ﾈ.ﾘｴ.ｺ..
00000050  CD 21 B4 4F CD 21 73 08-C6 06 00 00 00 90 EB 06   ﾍ!ｴOﾍ!s.ﾆ.......
00000060  C6 06 00 00 FF 90 07 1F-5D 5F 5E 5A 59 5B 58 CF   ﾆ.......]_^ZY[Xﾏ
00000070  D0 9C 24 00 C4 10 00 01-01 00 01 C4 17 00 01 01   ﾐ.$.ﾄ......ﾄ....
00000080  01 02 C4 2E 00 01 01 00-01 C4 3A 00 01 01 00 02   ..ﾄ......ﾄ:.....
00000090  C4 42 00 01 01 00 02 88-A0 09 00 01 00 02 FF 2A   ﾄB...... ......*
000000A0  2E 2A 00 D3 8A 02 00 00-74 00 00 00 00 00 00 00   .*.ﾓ....t.......
```

MASM TEMP;　⏎

これで，TEMP. OBJ（リスト3-17）というオブジェクトファイルができ上がります。ASM86のようにリスティングファイルなどを生成することも可能です。

MASM TEMP, TEMP, TEMP, TEMP　⏎

とすれば，

TEMP. OBJ

TEMP. LST（リスト3-18）

TEMP. CRF（リスト3-19）

が得られます。.LST はリスティングファイル，.CRF はクロスリファレンスファイルです。

また，最後に

/D, /O, /X

などのスイッチを使うことにより，8進出力にするとか条件が偽になった部分のリスティング抑制などMACRO86の動作を規定することができます。このように，デフォルトが .OBJ ファイル生成のみであるのはよく考えてあると思います。.LST ファイルや .CRF ファイルなどが必要になることはほとんどありませんから。

リスト3-18 リスティングファイルTEMP. LST

```
 The Microsoft MACRO Assembler              06-17-:0      PAGE    1-1


 1                                      ;
 2                                      ; SERCH FILE & DIRECTORY NAME
 3                                      ;
 4          = 0000                      ERROR    EQU     00H
 5          =-0001                      NORMAL   EQU     NOT      ERROR
 6
 7                                      ENTRY    MACRO
 8                                               PUSH    AX
 9                                               PUSH    BX
10                                               PUSH    CX
11                                               PUSH    DX
12                                               PUSH    SI
13                                               PUSH    DI
14                                               PUSH    BP
15                                               PUSH    DS
16                                               PUSH    ES
17
18                                               MOV     AX,CS
19                                               MOV     DS,AX
20                                               ENDM
```

```
21
22                                   EXIT    MACRO
23                                           LOCAL    L1,L2
24                                           JNC      L1
25                                           MOV      COND,ERROR
26                                           JMP      SHORT L2
27                                   L1:     MOV      COND,NORMAL
28                                   L2:
29                                           POP      ES
30                                           POP      DS
31                                           POP      BP
32                                           POP      DI
33                                           POP      SI
34                                           POP      DX
35                                           POP      CX
36                                           POP      BX
37                                           POP      AX
38                                           IRET
39
40                                           ENDM
41
42       0000                        CODE    SEGMENT  PUBLIC
43                                           ASSUME   CS:CODE,DS:CODE
44       0000                                ORG      0000H
45                                           ENTRY
46       0000   50                 +         PUSH     AX
47       0001   53                 +         PUSH     BX
48       0002   51                 +         PUSH     CX
49       0003   52                 +         PUSH     DX
50       0004   56                 +         PUSH     SI
51       0005   57                 +         PUSH     DI
52       0006   55                 +         PUSH     BP
53       0007   1E                 +         PUSH     DS

 The Microsoft MACRO Assembler            06-17-:0       PAGE     1-2


54       0008   06                 +         PUSH     ES
55       0009   8C C8              +         MOV      AX,CS
56       000B   8E D8              +         MOV      DS,AX
57
58       000D   B4 1A                        MOV      AH,1AH   ; SET DTA
59       000F   BA 0100 R                    MOV      DX,OFFSET DTA
60       0012   CD 21                        INT      21H
61
62       0014   B4 4E                        MOV      AH,4EH   ; FIND MATCH FI
                             LE
63       0016   BA 0201 R                    MOV      DX,OFFSET PATH_NAME
64       0019   B9 0017                      MOV      CX,17H
65       001C   CD 21                        INT      21H
66
67       001E                        NEXT:
68                                           ENTRY
69       001E   50                 +         PUSH     AX
70       001F   53                 +         PUSH     BX
71       0020   51                 +         PUSH     CX
72       0021   52                 +         PUSH     DX
73       0022   56                 +         PUSH     SI
74       0023   57                 +         PUSH     DI
75       0024   55                 +         PUSH     BP
76       0025   1E                 +         PUSH     DS
77       0026   06                 +         PUSH     ES
78       0027   8C C8              +         MOV      AX,CS
79       0029   8E D8              +         MOV      DS,AX
80
81       002B   B4 1A                        MOV      AH,1AH   ; SET DTA
82       002D   BA 0100 R                    MOV      DX,OFFSET DTA
83       0030   CD 21                        INT      21H
84
85       0032   B4 4F                        MOV      AH,4FH   ; STEP THROUGH
                             A DIRECTORY MATCHING FILES
```

```
86      0034  CD 21                              INT     21H
87
88                                               EXIT
89      0036  73 08                       +      JNC     ??0000
90      0038  C6 06 0200 R 00 90          +      MOV     COND,ERROR
91      003E  EB 06                       +      JMP     SHORT ??0001
92      0040  C6 06 0200 R FF 90          + ??0000: MOV  COND,NORMAL
93      0046                              + ??0001:
94      0046  07                          +      POP     ES
95      0047  1F                          +      POP     DS
96      0048  5D                          +      POP     BP
97      0049  5F                          +      POP     DI
98      004A  5E                          +      POP     SI
99      004B  5A                          +      POP     DX
100     004C  59                          +      POP     CX
101     004D  5B                          +      POP     BX
102     004E  58                          +      POP     AX
103     004F  CF                          +      IRET
104

 The Microsoft MACRO Assembler             06-17-:0      PAGE     1-3


105     0100                                     ORG     100H
106     0100      80 [                    DTA    DB      128 DUP(?)
107                       ??
108                            ]
109
110     0200                                     ORG     200H
111     0200  FF                          COND   DB      NORMAL
112     0201  2A 2F 2A 00                 PATH_NAME      DB      '*.*',0
113
114     0205                              CODE   ENDS
115                                              END


 The Microsoft MACRO Assembler             06-17-:0      PAGE     Symbols
                                 -1


Macros:

                N a m e                   Length

ENTRY. . . . . . . . . . . . . . .        0004
EXIT . . . . . . . . . . . . . . .        0005

Segments and groups:

                N a m e                   Size    align   combine class

CODE . . . . . . . . . . . . . . .        0205    PARA    PUBLIC

Symbols:

                N a m e                   Type    Value   Attr

COND . . . . . . . . . . . . . . .        L BYTE  0200    CODE
DTA. . . . . . . . . . . . . . . .        L BYTE  0100    CODE    Length =0080
ERROR. . . . . . . . . . . . . . .        Number  0000
NEXT . . . . . . . . . . . . . . .        L NEAR  001E    CODE
NORMAL . . . . . . . . . . . . . .        Number  - 0001
PATH_NAME. . . . . . . . . . . . .        L BYTE  0201    CODE
??0000 . . . . . . . . . . . . . .        L NEAR  0040    CODE
??0001 . . . . . . . . . . . . . .        L NEAR  0046    CODE

Warning Severe
Errors  Errors
0       0
```

リスト3-19 クロスリファレンスファイルTEMP. CRFの内容

```
Dump Version 2.0

00000000  3A 50 07 04 04 04 02 45-52 52 4F 52 04 02 4E 4F   :P.....ERROR..NO
00000010  52 4D 41 4C 01 45 52 52-4F 52 04 04 04 04 04 04   RMAL.ERROR......
00000020  04 04 04 04 04 04 04 04-04 04 04 04 02 45 58 49   .............EXI
00000030  54 04 04 04 04 04 04 04-04 04 04 04 04 04 04 04   T...............
00000040  04 04 04 04 02 43 4F 44-45 04 01 43 4F 44 45 01   .....CODE..CODE.
00000050  43 4F 44 45 04 04 01 45-4E 54 52 59 04 04 04 04   CODE...ENTRY....
00000060  04 04 04 04 04 04 04 04-04 04 01 44 54 41 04 04   ...........DTA..
00000070  04 04 01 50 41 54 48 5F-4E 41 4D 45 04 04 04 04   ...PATH_NAME....
00000080  02 4E 45 58 54 04 01 45-4E 54 52 59 04 04 04 04   .NEXT..ENTRY....
00000090  04 04 04 04 04 04 04 04-04 04 01 44 54 41 04 04   ...........DTA..
000000A0  04 04 04 04 01 45 58 49-54 04 01 3F 3F 30 30 30   .....EXIT..??000
000000B0  30 04 01 43 4F 4E 44 01-45 52 52 4F 52 04 01 3F   0..COND.ERROR..?
000000C0  3F 30 30 30 31 04 02 3F-3F 30 30 30 30 01 43 4F   ?0001..??0000.CO
000000D0  4E 44 01 4E 4F 52 4D 41-4C 04 02 3F 3F 30 30 30   ND.NORMAL..??000
000000E0  31 04 04 04 04 04 04 04-04 04 04 04 04 04 02 44   1..............D
000000F0  54 41 04 04 04 04 04 02-43 4F 4E 44 01 4E 4F 52   TA......COND.NOR
00000100  4D 41 4C 04 02 50 41 54-48 5F 4E 41 4D 45 04 04   MAL..PATH_NAME..
00000110  01 43 4F 44 45 04 04 05-00 00 00 00 00 00 00 00   .CODE..........
```

MACRO86ソースプログラムの特徴

マクロ機能　MACRO86というくらいですから，マクロ機能は最大の特徴の１つです。

プログラムを組んでいると同じようなテキストが何度も出てくることがあります。ASM86などではそのたびに同じようなテキストを書かなければなりませんが，マクロ機能があれば一度定義するだけで，あとは同じテキストを書く代わりにマクロ名を置けばよいのです。

たとえば，レジスタ AX～DX をプッシュするとします。マクロ機能がなければ毎回，

```
PUSH AX  ┐
PUSH BX  ├──────────①
PUSH CX  │
PUSH DX  ┘
```

と書かなければなりません。しかし，マクロ機能があれば

```
PUSH_ALL  MACRO
          PUSH AX
          PUSH BX
          PUSH CX
```

```
            PUSH DX
            ENDM
```
と PUSH_ALL を一度定義するだけで，必要なときに

```
   PUSH_ALL
```
と書くだけで①を記述したのと同じ働きをします。また，引数を指定することによって展開する命令を変えることもできます。変数Xに AX レジスタを介して値をロードする命令 LET_X を作ってみましょう。

```
   MACRO  LET_X  PAR
          MOV    AX,PAR
          MOV    X,AX
          ENDM
```
　これを使うときに

```
   LET_X  123
```
とすれば

```
   MOV  AX, 123
   MOV  X, AX
```
となりますし，

```
   LET_X  BX
```
とすれば

```
   MOV  AX, BX
   MOV  X, AX
```
と展開されます。

　マクロ機能をうまく使えば FOR～NEXT 文や WHILE～WEND 文，PRINT 文などを作ることもできます。アセンブリ言語をほとんど BASIC のように書くことも可能なわけです。

　マクロ機能は便利ですが，使いすぎるとアセンブル時間が長くなります。また，よく確かめたマクロならばよいのですが，バグがあるマクロの場合はデバッグが困難になります。マクロ機能を利用するときはよく使うものをライブラリのようにして持っておくのがよいでしょう。やむをえずその場で作る場合は，よく正統性を確かめてから使うことをお勧めします。

リロケータブル　　MACRO86はリロケータブルなオブジェクトを生成しま

す。サブルーチンごとにファイルを分けて個別にアセンブルし，最後にリンクして1つのプログラムにすることができます。また，よく使うサブルーチンをライブラリに登録し，必要なときだけリンクすることも可能です。さらに，ほかの言語とリンクすることもでき，メインプログラムはC言語，グラフィックスはアセンブリ言語でというプログラムも書けます。

アセンブリ言語で大きなプログラムを作るとき，1つのファイルで作るとアセンブルに大変な時間がかかりますが，小さなサブルーチンに分けて個別にアセンブルすればアセンブル時間は短くてすみます。

豊富なディレクティブ　MACRO86には次の多くの**ディレクティブ**があります(表3-3)。

メモリディレクティブはメモリ空間を制御する，条件ディレクティブは条件アセンブルのための，そしてマクロディレクティブはマクロ定義用の擬似命令です。リスティングディレクティブは，LST ファイルの形式を決めます。

MACRO86私見

プログラムを組むのは簡単だが，デバッグが大変というのが正直な感想です。デバッガはDEBUG. COMを使うわけですが，これはDDT86. CMDと同程度でしかなく，一度シンボリックデバッグを知ってしまうとどうにもまどろっこしくて困ります。

MS-DOS用のシンボリックデバッガもありますが，PUBLIC宣言した変数やラベルしか表示できないためほとんど使いものになりません。これはMACRO86がシンボルファイルを生成しないためといえるでしょう。とはいえ，MS-DOSで高級言語とリンク可能なアセンブラといえばMACRO86以外考えられないでしょう。デバッグのことを考えながら注意深くコーディングすれば，これほど高級なアセンブラはないといえるかもしれません。

WACS

DISK BASICから呼ぶ機械語サブルーチン作成用に作られたのが，このWACSです。これまで，DISK BASICで機械語サブルーチンを組むとき，短い場合はモニタのアセンブラで，長い場合ASM86かMACRO86で組んでファイル転送する方法が普通でした。モニタのアセンブラではラベルの処理が大変

表3-3

メモリディレクティブ	
ASSUME	セグメントレジスタの指定
COMMENT	コメントの開始
DB, DW, DD, DQ, DT	変数定義，初期化
END	終了
EQU	値の割りつけ
=	シンボルのセット，再定義
EVEN	ロケーションカウンタの偶数化
EXTRN	外部のシンボル
GROUP	セグメントのグループ化
INCLUDE	ソースコードの挿入
LABEL	ラベルづけ
NAME	名前づけ
ORG	ロケーションカウンタのセット
PROC	プロシージャ
PUBLIC	外部から参照可能化
RADIX	基数変換
RECORD	レコード定義
SEGMENT	セグメント定義
STRUC	構造体
条件ディレクティブ	
IF××××	
ELSE	
ENDIF	
マクロディレクティブ	
MACRO	マクロ定義
ENDM	マクロの終了
EXITM	マクロからの抜け出し
LOCAL	局所ラベル
PURGE	マクロの削除
REPT	繰り返し
IRP	不定回繰り返し
IRPC	文字の繰り返し
&, 〈 〉, ; : , ! , %	
リスティングディレクティブ	
PAGE	ページの幅，長さ
TITLE	タイトル
SUBTTL	サブタイトル
%OUT	メッセージ出力
.LIST, .XLIST	リスト出力の抑止
.SFCOND/.LFCOND/.TFCOND	真偽によるリスト出力，抑止
.XALL/.LALL/.SALL	マクロ部分のリスト出力，抑止
.CREF/.XCREF	

で，ASM86やMACRO86では別のOSを立ち上げてファイル転送しなければならず，やはり大変でした。ファイルコンバータを自分で作った人も多いでしょう。

ところが，WACSはBASIC上で動作します。そのためOSの立ち上げやファイル転送の必要はありません。以下，WACSの特徴を述べます。

超高速アセンブル

ASM86もMACRO86もソースファイルはディスク上にあります。そのため，アセンブル時間の多くがディスクアクセスに費やされます。ところが，WACSはプログラムをメモリ上にもっているため，非常に高速なアセンブルが可能です。短いプログラムならばあっという間に，少々長くても数秒でアセンブルを終了します。ASM86やMACRO86では5分10分待つことはざらですが，WACSはそんなことはありません。

半面，大きなプログラムはメモリに入りきらないこともあります。しかし，その場合もコモンラベルを使い，プログラムを分割してアセンブルできるわけです。BASICから呼ぶ機械語サブルーチンがそんなに巨大になることはあまりないと思いますが……。

スクリーンエディタと一体型

WACSにはスクリーンエディタがついています。しかし，DOS環境のようにエディタを呼び出してテキストを作り，エディタを抜けてアセンブラを呼び出し，アセンブルして実行する「エディタはエディタ，アセンブラはアセンブラ」というバラバラな状態ではありません。WACSではエディタやアセンブラが有機的に結合しています。

エディタでテキストを作成すれば，エディタを抜けることなく，すぐにアセンブルを開始します。また，エラーがあればすぐにエディットしてふたたびアセンブルできます。

WACSのエディタはマルチウィンドウで，スクリーンエディタとしても十分すぎる機能をもっています。また，スクロールもきわめて高速でコンパイラで書かれたエディタのようにイライラさせません。

エラートレース機能

アセンブル中にエラーが生じた行にはエラーマークがつけられ，WACSのエディタはそのエラー行を瞬時に探し出します。一般のアセンブラを使っている場合，エラー行探しは面倒な作業ですが，WACSでは非常に簡単です。

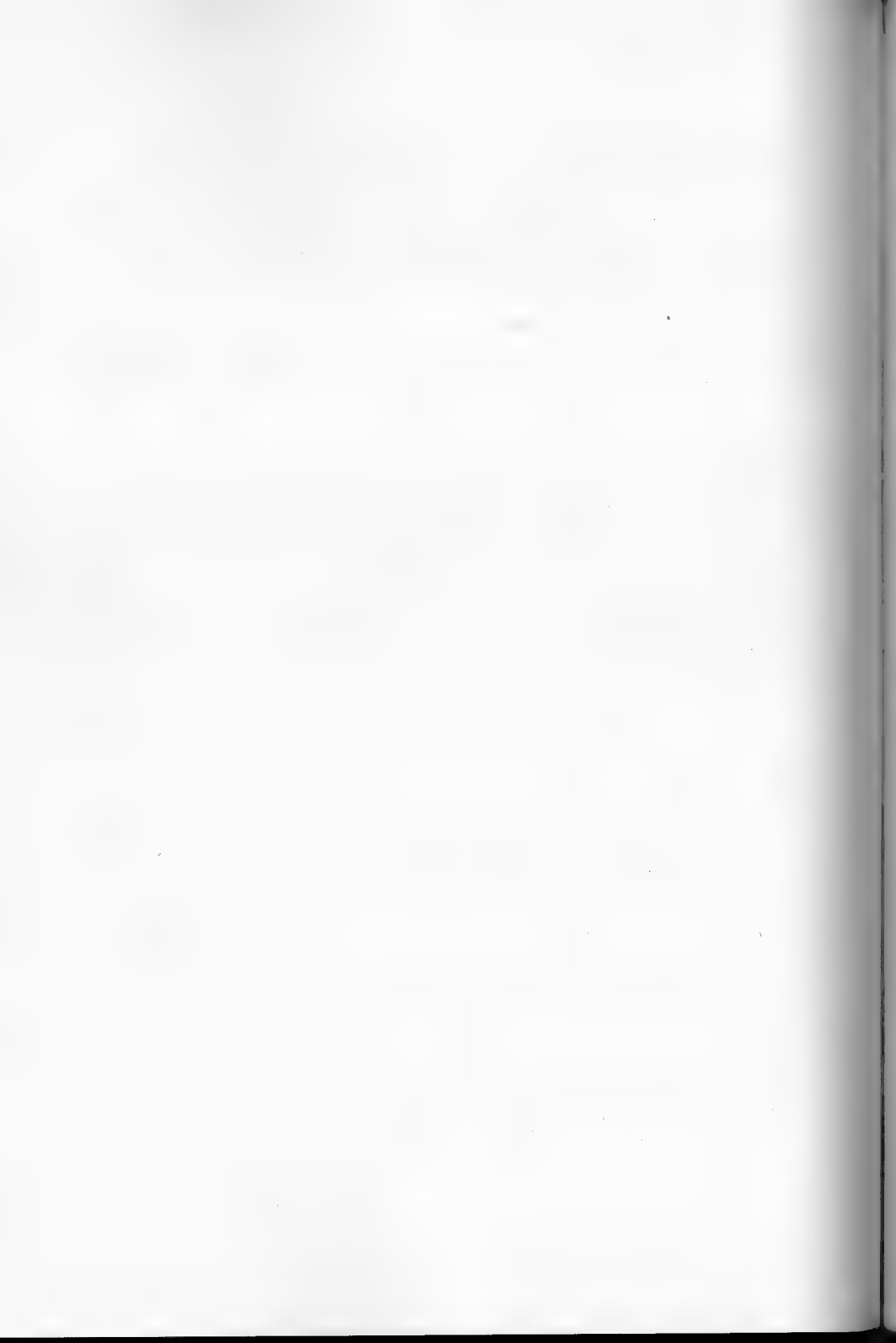
行シンタックスチェック機能

1行シンタックスチェック機能を使うと，1行を打ち終わった段階でシンタックスチェックを行いますから，打ち間違いはなくなります。

BASコマンド

機械語ルーチンが完成したら，BASICとリンクが必要です。機械語ルーチンのエントリが1つならよいのですが，複数あったり多くの機械語ルーチンの変数とやりとりする場合，ラベルや変数の値を調べるのはやっかいです。DOSのアセンブラの場合シンボルテーブを見ながらBASICの変数に代入していきますが，WACSはBASコマンドでリンクに必要なBASICプログラムを自動作成してくれます。

WACSはマニアの作ったマニアのためのアセンブラといえるのではないでしょうか。これまでのエディタ・アセンブラとは違った使いやすいアセンブラでしょう。

第4章

PC-9801のグラフィック

GDCのプログラミング

アセンブラの使い方が分かったところで，PC-9801で重要なGDC (7220)のプログラミングを解説します。

CPU8086とGDC7220

8ビット系パソコンをかなり使ってきましたが，どれもグラフィックや速度の面で満足できないものばかりでした。16ビット系にしてもCPUでラインを描いているものが大半で，グラフィックに関してはどれもかんばしくないといった印象をもっています。

その中でPC-9801はCPUに8086，グラフィックにGDC (Graphic Display Controller) NEC μPD7220Dを使っており，ほぼ満足のいくパソコンといえるでしょう。

もちろん不満な点はあります。たとえば8086は16ビットというものの，かなり以前に発表された石でアーキテクチャはあまりよいとは思えません。特にセグメントの概念が好きになれません。

後出の68000と比較するのは酷かもしれませんが，アセンブラでプログラミングしたとき，68000はあたかも高級言語であるかのように思えました。同じ16ビットでこうも違うのかと感じてしまいます。しかし，これも16ビットCPUの8088に比べれば光っているといえるでしょう。

一方，GDC7220は1981年の春に発表されて以来，注目を集めてきたLSIです。直線，円弧，ズーム，パニングなど豊富な機能をもっています。

もちろんこの石にも欠点はあります。その1つに外づけICをかなり必要とすることがあげられます。さらにパラメータの設定が多すぎることも欠点といえましょう。私自身，68000やZ80のアセンブラ，FORTH，BASICなどでドライバルーチンを組んでみましたが，確かにパラメータが多く，たった1本のラインを引くのにどうしてこんなに長いプログラムが必要なのかと思ってしまうほどです。後出のリスト4−1を参照してください。

しかし，画面は非常に高速で，パラメータセットのオーバーヘッドを相殺しても十分余りあります。インテルがセカンドソースを出したり，シャープがパ

ソコンに採用するのもうなずけます。

ズーム機能もなかなかおもしろく，しばらくは7220の機能を中心に述べてみましょう。

BASIC

BASICは例によって N88(86)-BASIC がついています。機能的には申し分ないのですが，せっかくの 8086+GDC マシンも BASICではありがたみが半減してしまいます。

ワールド座標の概念は，便利な半面ウィンドウィングなどに時間がかかります。また，7220のサークル機能も使えません。さらに，1ドットずつ打っていくグラフィックなど，ハードウェアの能力を犠牲にしているところも多くあります。

しかし，この PC-9801 はアセンブラで動かすとすごい力を発揮するでしょう。もちろん BASICマシンとしても，パソコンの中では群を抜いています。

Lineのプログラム

ここでは，BASICでLINE文を使わずに線を引くプログラム（リスト4-1）をとりあげました。BASICで直接7220をドライブしてみました。

アセンブラで画面処理をするとき，GDCを操作することは不可欠ですが，コマンド，パラメータの設定など分かりにくい部分が多いので，ここではBASICで書いてみました。あとでアセンブラによる例も載せ，ゲームを作るときの参考にしていただくつもりです。

GDCのコントロールは，コマンドとパラメータを送ることによって行います。コマンドライトポートは A2H，パラメータライトポートは A0H です。また，描画中だとかFIFOがFULLだとか，GDCの状態を示すステータスリードポートはA0Hです。VRAMデータを読むデータリードポートにはA2Hが充てられます。

GDCコマンドには，表4-1の21種類があります。個々のコマンドについては，後ほどプログラムを追いながら説明していきます。

コマンドの送り方ですが，無条件に書けばよいというわけではありません。GDCには，コマンドやパラメータを16バイトまでためておくFIFOがありま

す。これでCPUがGDCにデータを送り終わると，すぐにほかの仕事ができるようになります。ちょうど，プリンタのリングバッファの働きに似ています。

そのため，FIFOがいっぱいになっていないことを書き込む前に確認しなければなりません。逆に空になっていることを調べてもよいのですが，FIFOが空になるまで待たされるため，前者の方法がよいでしょう。

先ほど，ステータス・リードポートはA0Hといいましたが，各ビットは表4-2のようになっています。FIFO FULLは第1ビットですから，

[inp(A0H)and2]＝0

を確認して，その後にコマンドなどを送ればよいのです（後出OUT．COMMAND の項参照)。

ラインを1本引くのに必要なコマンドは TEXTW，WRITE，CSRW，

表4-1　NEC μPD7220 GDCユーザーズマニュアルより引用

GDCには以下に示す21種のコマンドが用意されている。

動作制御

コマンド	動作内容	(MSB)コマンド・コード(LSB)							
RESET	初期化動作	0	0	0	0	0	0	0	0
SYNC	動作モード，同期信号波形の定義	0	0	0	0	1	1	1	DE
MASTER/SLAVE	マスタ動作，スレーブ動作の選択	0	1	1	0	1	1	1	M

表示制御

コマンド	動作内容	(MSB)コマンド・コード(LSB)							
START**	表示の開始の指示	0	1	1	0	1	0	1	1
		0	0	0	0	1	1	0	1
STOP	表示の停止の指示	0	0	0	0	1	1	0	0
ZOOM	拡大表示係数，拡大描画係数の設定	0	1	0	0	0	1	1	0
SCROLL	表示開始アドレス，表示領域の設定	0	1	1	1	←RA→			
CSRFORM	文字表示時のカーサ形状などの設定	0	1	0	0	1	0	1	1
PITCH	映像メモリ水平方向ワード数の設定	0	1	0	0	0	1	1	1
LPEN	ライトペン・アドレスの読み出し指示	1	1	0	0	0	0	0	0

VECTW，VECTE です。

TEXTW

このコマンドはLINE，CIRCLE などの線を実線にするか，破線にするかを指定します。

LINE の場合，78H で，パラメータは２つです。パラメータを FFH，FFH とすると実線，33H，33H とすると破線が描かれます。また，00H，00H とすればラインを消去することができます。どれも BASIC の LINE 命令の感覚で使えます。

ためしにリスト1240行の

表 4-2

bit 0………DATA READY
bit 1………FIFO FULL
bit 2………FIFO EMPTY
bit 3………DRAWING
bit 4………DMA EXECUTE
bit 5………VSYNC
bit 6………HBLANK
bit 7………LIGHT PEN

描画制御

コマンド	動作内容	(MSB)コマンド・コード(LSB)							
VECTW	描画に必要な各種パラメータの設定	0	1	0	0	1	1	0	0
VECTE	直線，四辺形，円弧描画の実行の指示	0	1	1	0	1	1	0	0
TEXTW	グラフィックス・テキスト・コード設定	0	1	1	1	1	←RA→		
TEXTE	グラフィックス・テキスト描画実行指示	0	1	1	0	1	0	0	0
CSRW	描画アドレスの設定	0	1	0	0	1	0	0	1
CSRR	描画アドレスの読み出しの指示	1	1	1	0	0	0	0	0
MASK	マスク・レジスタ値の設定	0	1	0	0	1	0	1	0

映像メモリ制御

コマンド	動作内容	(MSB)コマンド・コード(LSB)					
WRITE	パラメータの映像メモリへの書き込み準備	0	0	1	WLH	0	MOD
READ	映像メモリ・データの読み出しの指示	1	0	1	WLH	0	MOD
DMAW	映像メモリへのDMA転送開始の指示	0	0	1	WLH	1	MOD
DMAR	映像メモリからのDMA転送開始の指示	1	0	1	WLH	1	MOD

PARAMETER＝&HFF
を &H66 にしてみましょう。破線になるはずです。

WRITE

WRITE コマンドは，ラインを引く場合 20H を，消去したいときには 22H を用います。

CSRW

描画を開始する点のアドレスと，ドットアドレスを設定するコマンドです。

これを使うにあたっては，VRAM のアドレスが CPU と GDC とではまったく違うことに注意します。GDC のアドレスはブルーが 4000H から，レッドが 8000H から，グリーンが C000H からです（図 4－1）。

ドットアドレスについても同様の注意が必要です。CPU では左が MSB なのに対し，GDC では右が MSB となっています（図 4－2）。

また，CPU は 1 アドレス 8 ビットなのに対し，GDC は16ビットになってい

図4-1

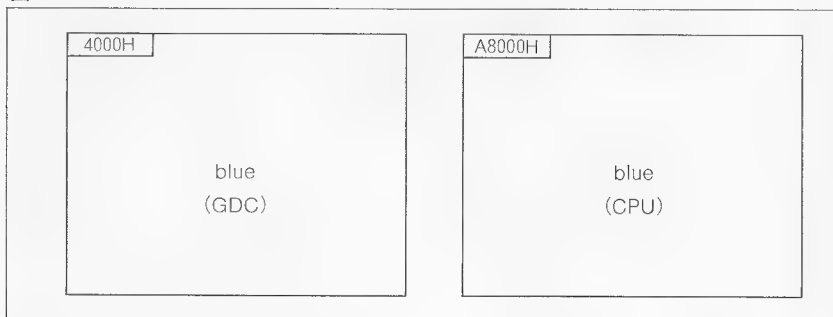
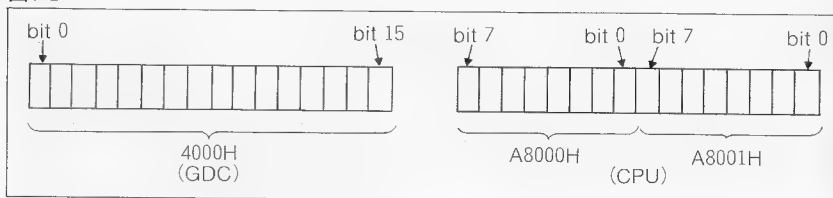



図4-2

ます。アドレスの割り振りこそ違いますが，CPU，GDC どちらでアクセスしようと物理的にはまったく同じ VRAM です。

ワークアドレスを EAD, ドットアドレスを dAD とすると，ブルーの点（x, y）は，

EAD＝&H4000＋40*y＋int(x/16)

dAD＝X mod 16

で求めることができます。

CSRW コマンドのアドレスは 49H で，パラメータは EAD_L，EAD_M，それに

dAD	0 0	EAD_{11}

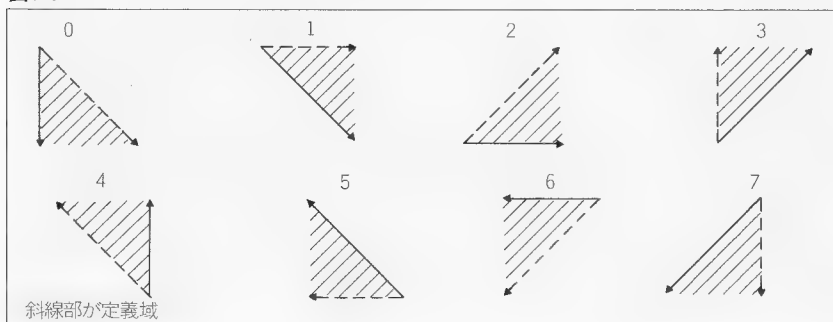
の3つを表せばよいのです。これについてはプログラム中の ＊CULADR を参照してください。

VECTW

描画の方向，描画用パラメータなどを設定するコマンドで，アドレスは 4CH です。

GDC は基本的に0°～45°方向の直線しか引けません。それを DIR（方向）を変えることで全方向引けるようにしましょう（図4－3）。DIR は45°おきの8方向が存在します。ただし，ラインの場合データを入れ替えることによって4方向で処理できます。

LINE$(x_1, y_1)-(x_2, y_2)$

図4-3

のとき，

$\Delta x = x_2 - x_1$，$\Delta y = y_2 - y_1$

から

$\Delta x \geqq 0$

つまり，(x_1, y_1) と (x_2, y_2) を入れ替えればよいのです。

DIR の決定と Δx，Δy の値との関係は図 4－4 のフローチャートを見れば分かるでしょう。この部分はプログラムの＊CULDIR に当たります。

直線を引くには GDC 内部のレジスタ DC，D，D2，D1 を設定しなければなりません。

設定値は

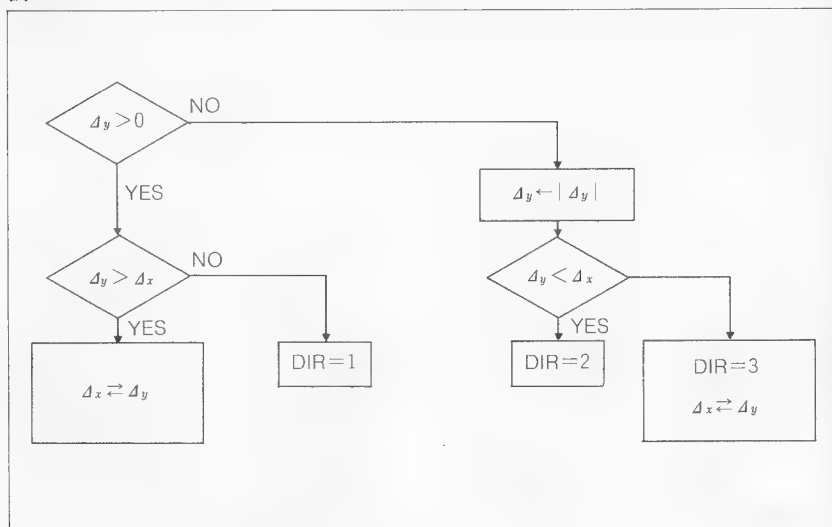
DC＝Δx

D＝2＊Δy－Δx

D2＝2＊Δy－2＊Δx

D1＝2＊Δy

で，プログラムでは＊CULPAR の部分です。結局 VECTW はコマンドとして 4CH を送ったあと，パラメータ

図4-4

8+DIR, DC_L, DC_H, D_2, D_H, $D2_L$, $D2_H$, $D1_L$, $D1_H$

を送ればよいのです（VECTW の項参照）。

VECTE

GDC に描画するように命令を出すコマンドで，アドレスは60CH です。

以上，いくつかのコマンドを紹介しましたが，これらを踏まえて作られたのがプログラム4-1です。

リスト4-1


```
1000 *LINE.ROUTINE  'line(x1,y1)-(x2,y2)
1010 GOSUB *INPUT.PARAMETER
1020 GOSUB *TEXTW
1030 GOSUB *WRITE1
1040 GOSUB *INIT
1050 GOSUB *CULADR
1060 GOSUB *CSRW
1070 GOSUB *CULDIR
1080 GOSUB *CULPAR
1090 GOSUB *VECTW
1100 GOSUB *VECTE
1110 END
1120 *OUT.COMMAND WHILE INP(&HA0) AND 2:WEND:OUT &HA2,COMMAND:RETURN
1130 *OUT.PARAMETER WHILE INP(&HA0) AND 2:WEND:OUT &HA0,PARAMETER:RETURN
1140 *OUT.WORD.PARAMETER
1150 PL=VAL("&H"+RIGHT$(RIGHT$("0000"+HEX$(PARAMETER),4),2))
1160 PH=VAL("&H"+LEFT$(RIGHT$("0000"+HEX$(PARAMETER),4),2))
1170 PARAMETER=PL : GOSUB *OUT.PARAMETER
1180 PARAMETER=PH : GOSUB *OUT.PARAMETER
1190 RETURN
1200 *INPUT.PARAMETER:INPUT "X1,Y1,X2,Y2";X1,Y1,X2,Y2:RETURN
1210 '
1220 *TEXTW
1230 COMMAND=&H78:GOSUB *OUT.COMMAND
1240 PARAMETER=&HFF:GOSUB *OUT.PARAMETER:GOSUB *OUT.PARAMETER
1250 RETURN
1260 *WRITE1 COMMAND=&H20:GOSUB *OUT.COMMAND:RETURN
1270 *INIT
1280  IF X1>X2 THEN SWAP X1,X2:SWAP Y1,Y2
1290  IF X1=X2 THEN IF Y1>Y2 THEN SWAP Y1,Y2
1300 RETURN
1310 *CULADR
1320  EAD=&H4000+40*Y1+X1¥16
1330  EADM=EAD¥256:EADL=EAD MOD 256
1340 DAD=X1 MOD 16
1350 RETURN
1360 *CSRW
1370  COMMAND=&H49:GOSUB *OUT.COMMAND
1380  PARAMETER=EADL : GOSUB *OUT.PARAMETER
1390  PARAMETER=EADM:GOSUB *OUT.PARAMETER
1400  PARAMETER=16*DAD:GOSUB *OUT.PARAMETER
1410 RETURN
1420 *CULDIR
1430  DX=X2-X1:DY=Y2-Y1
1440  IF DY<=0 THEN *DY.LE.0
1450  IF DY>0 THEN IF DY>DX THEN DIR=0 :SWAP DX,DY:GOTO *END.CULDIR ELSE DIR=1:G
```

```
OTO *END.CULDIR
1460 *DY.LE.0 DY=-DY:IF DY<DX THEN DIR=2 ELSE DIR=3:SWAP DX,DY
1470 *END.CULDIR
1480 RETURN
1490 *CULPAR DC=DX:D=2*DY-DX:D2=2*DY-2*DX:D1=2*DY:RETURN
1500 *VECTW
1510  COMMAND=&H4C:GOSUB *OUT.COMMAND
1520 PARAMETER=8+DIR:GOSUB *OUT.PARAMETER
1530 PARAMETER=DC:GOSUB *OUT.WORD.PARAMETER
1540 PARAMETER=D:GOSUB *OUT.WORD.PARAMETER
1550 PARAMETER=D2:GOSUB *OUT.WORD.PARAMETER
1560 PARAMETER=D1:GOSUB *OUT.WORD.PARAMETER
1570 RETURN
1580 *VECTE COMMAND=&H6C:GOSUB *OUT.COMMAND:RETURN
```

ROM内ルーチンについて

PC-9801のROM内にはCP/MのBIOSのようなものがあって，これをコールすることによって，かなりのサービスを受けられます。1文字入出力，ブザー，画面関係など，およそ必要なルーチンは，ROM内にあります。コールの方法は，ソフトウェアインタラプト（INT）命令で，レジスタを引数として行っているようです。

ここでとり上げるLineルーチンも，このBIOSのような部分にあります。ですから，それをコールすることにより，Lineを引くことも可能となります。しかし，それは汎用ルーチンであり，またソフトウェアインタラプトによりコールするため，オーバーヘッドが大きいのです。ゲームや高速描画を必要とする場合，やはり自分でLineルーチンを書かなければなりません。初期設定など高速性の要求されないときには，ROM内ルーチンを大いに利用してほしいと思います（CP/M86の管理下でも，このROM内ルーチンは利用可能です）。

Lineルーチン

それでは，Lineルーチンの説明に入りましょう。前のBASICによるプログラムを，アセンブラに落としていくことにします。アセンブラでプログラムする前に，高級言語でロジックチェックして，確実に動くことを確認してから，アセンブラに落としていくのは，きわめて効率がよく，ロジカルなバグの発生を非常に低く抑えることができます。高級言語で，ここはどんなふうにアセンブルするかを考えながら，プログラムしていくと効果的です。

このルーチンは説明用ですから，最適化（サブルーチンを展開したり，乗算をシフトで行ったり，引数をレジスタで渡したりなど）はしてはいません。も

っとも，速度がそれほど落ちるわけではありません。

それでは，プログラムを見ていきましょう。

Lineルーチンの使い方

x_1, y_1, x_2, y_2, color の各変数を，セットしてコールします。color は，

青 color＝0　　赤 color＝1　　緑 color＝2

とします。ほかの色の Line を引くときは，色を変えて，２度（白なら３度）Line ルーチンを呼びます。たとえば，紫は，赤と緑だから，

```
mov color, 1
call line
mov color, 2
call line
```

とします。

```
LINE:
        CALL    TEXTW       線種(実線破線等)を指定するルーチン
        CALL    WRITE       描画か消去かの指定するルーチン
        CALL    INIT        (x1, y1), (x2, y2)のイニシャライズ
        CAL     CULADR      描画実行アドレス,ドットアドレスをセットするルーチン
        CALL    CSRW        csrwコマンドの発行
        CALL    CULDIR      描画方向の計算
        CALL    CULPAR      vectw用パラメータのセット
        CALL    VECTW       vectwコマンド発行
        CALL    VECTE       vecteコマンド発行
        RET                 おわり
```

次に各サブルーチンを説明していきます。

textwルーチン

実線か破線かを指定します。

```
TEXTW:  MOV     COMMAND,78H
```

変数 command に 78h を代入します。このように，8086では，イミディエイト値をメモリに直接（レジスタを介さず）ロードできます。

```
        CALL    OUTC            GDCへコマンドを送出
        MOV     PARMTR,0FFH     パラメータをFFHにセット
        CALL    OUTP            GDCへパラメータを送出
        CALL    OUTP                    //
        RET                     おわり
```

writeルーチン

DGC に，描画か，消去かを指令するルーチンです。

```
WRITE:                                 描画を指定
        MOV     COMMAND,20H            DGCへコマンド送出
        CALL    OUTC                   おわり
        RET
```

initルーチン

(x_1, y_1) (x_2, y_2) を必要に応じて入れ替えるルーチンです。

```
INIT:

        MOV     AX,X2   }
        CMP     AX,X1   } もしx2>=x1ならばinit 1へ
        JGE     INIT1   }

        XCHG    X1,AX
```

8086では，レジスタとメモリを交換できます。

```
        MOV     X2,AX     x1とx2を交換
        MOV     AX,Y1   }
        XCHG    Y2,AX   } y1とy2を交換
        MOV     Y1,AX   }
INIT1:  MOV     AX,X1   }
        CMP     AX,X2   } x1≒x2ならばinit 2へ
        JNE     INIT2   }

        MOV     AX,Y2

        CMP     AX,Y1   }
        JGE     INIT3   } もしy2>=y1ならばinit 3へ

        XCHG    Y1,AX   }
        MOV     Y2,AX   } y1⇄y2
        RET               おわり
```

culadr ルーチン

GDC の描画開始アドレス，ドットアドレスを計算するルーチンです。

```
CULADR:
        MOV     AL,COLOR   }
        CMP     AL,0       } Color≒0ならば0 Cul 1へ
        JNE     CUL1       }

        MOV     BX,4000H   }
                           } color=0ならばVRAMの開始番地4000Hを持ってcul 2へ
        JMPS    CUL2       }

CUL1:   CMP     AL,1       }
        JNE     CUL3       } color≒1ならばcul 3へ

        MOV     BX,8000H   }
        JMPS    CUL2       } color=1ならばVRAM開始番地8000Hを持ってcul 2へ

        MOV     BX,0C000H    color=2ならばVRAM開始番地COOOH
CUL2:   MOV     EAD,BX       eadに一時bxを入れる
```

```
MOV     AX,X1      }
MOV     CL,4       } x1/16を計算
SHR     AX,CL      }
```

このように，8086ではシフト回数を CL レジスタで指定できます。

```
ADD     EAD,AX
```

メモリにレジスタ値を加えることも可能です。

```
MOV     AX,40      }
MOV     BX,Y1      } 40 * y1を計算
MUL     BX         }
```

8086には，乗算命令（mul, imul），除算命令（div, idiv）があり，プログラムの高速化，簡略化が図られています。

```
ADD     EAD,AX       eadに代入

MOV     AX,X1      }
AND     AL,0FH     } dad = x1 mod 16
MOV     DAD,AL     }
RET                  おわり
```

csrwルーチン

CSRW コマンドを発行するルーチンです。

```
CSRW:
        MOV     COMMAND,49H  } GDCに49h（：csrw)を送出
        CALL    OUTC

        MOV     AX,EAD       }
        MOV     WRDPAR,AX    } ワード長のデータeadをパロメータとしてGDCへ送出
        CALL    OUTWP        }

        MOV     AL,DAD       }
        MOV     CL,4         } al = dad * 16
        SHL     AL,CL        }

        MOV     PARMTR,AL    } dad * 16をパラメータとして送出
        CALL    OUTP         }
        RET                    おわり
```

culdirルーチン

描画方向，Δx，Δy を求めるルーチンです。

```
CULDIR:
        MOV     AX,X2        }
        SUB     AX,X1        } Δx = x2 − x1
        MOV     DELTA_X,AX   }

        MOV     AX,Y2        }
        SUB     AX,Y1        } Δy = y2 − y1
        MOV     DELTA_Y,AX   }
```

```
        CMP     AX,0            } ⊿y<=0ならばdir 1へ
        JLE     DIR1

        MOV     BX,DELTA_X      }
        CMP     AX,BX           } ⊿x<=⊿yならばdir 2へ
        JLE     DIR2            }

        MOV     DIR,0           dir=0

        MOV     AX,DELTA_X      }
        XCHG    DELTA_Y,AX      } ⊿x⇄⊿y
        MOV     DELTA_X,AX      }

        JMPS    DIR3            dir 3へjmpsはセクメント内ジャンプ

DIR2:   MOV     DIR,1           } dr=1としてdir 3へ
        JMPS    DIR3            }

DIR1:   NEG     DELTA_Y         ⊿y←-⊿y
        MOV     BX,DELTA_Y      }
                                } ⊿y>⊿xならばdir 4へ
        CMP     BX,DELTA_X      }
        JG      DIR4

        MOV     DIR,2           } dir=2としてdir 3へ
        JMPS    DIR3            }

DIR4:   MOV     DIR,3           dir=3

        MOV     AX,DELTA_X      }
        XCHG    DELTA_Y,AX      } ⊿x⇄⊿y
        MOV     DELTA_X,AX      }

DIR3:
        RET                     おわり
```

culparルーチン

vectw コマンド用パラメータの計算です。

```
CULPAR:
        MOV     AX,DELTA_X      } DC=⊿x
        MOV     DC_REG,AX       }

        MOV     AX,DELTA_Y      }
        SAL     AX,1            } D=2 *⊿y-⊿x
        SUB     AX,DELTA_X      }
        MOV     D_REG,AX        }

        MOV     AX,DELTA_Y      }
        SUB     AX,DELTA_X      } D2=2⊿y-2⊿x
        SAL     AX,1            }
        MOV     D2_REG,AX       }

        MOV     AX,DELTA_Y      }
        SAL     AX,1            } D1=2⊿y
        MOV     D1_REG,AX       }
        RET                     おわり
```

vectwルーチン

GDC へ vectw コマンドを発行します。

```
VECTW:
```

```
        MOV     COMMAND,4CH     } 4ch (vectw)をGDCへ送出
        CALL    OUTC

        MOV     AL,DIR          }
        OR      AL,8            } (dir or 8)をパラメータとして送出
        MOV     PARMTR,AL       }
        CALL    OUTP            }

        MOV     AX,DC_REG       }
        MOV     WRDAR,AX        } dc__regの値をパラメータとして発出
        CALL    OUTWP           }

        MOV     AX,D_REG        }
        MOV     WRDPAR,AX       } d__regの値をパラメータとして送出
        CALL    OUTWP           }

        MOV     AX,D2_REG       }
        MOV     WRDPAR,AX       } d2__regの値をパラメータとして送出
        CALL    OUTWP           }

        MOV     AX,D1_REG       }
        MOV     WRDPAR,AX       } d1__regの値をパラメータとして送出
        CALL    OUTWP           }
        RET                       おわり
```

vecteルーチン

vecte コマンドの発行するルーチンです。

```
VECTE:
        MOV     COMMAND,6CH     } 6chをコマンド送出
        CALL    OUTC            }

        RET                       おわり
```

outcルーチン

GDC にコマンドを送出するルーチンです。

```
OUTC:

OUTCLP: IN      AL,STATPT         (ステータスリードポートよりGDCのステータスを入力)
        AND     AL,2            }
        JNZ     OUTCLP          } GDCのFIFOがFullでなくなるまでまつ

        MOV     AL,COMMAND      }
        OUT     COUTPT,AL       } コマンドをコマンド出力ポートへ出力

        RET                       おわり
```

outpルーチン

GDC に,パラメータを送出するルーチンです。

```
OUTP:

OUTPLP: IN      AL,STATPT
```

```
        AND     AL,2                  } FIFOがFullでなくなるまでまつ
        JNZ     OUTPLP

        MOV     AL,PARMTR             } parmtrをパラメータ出力ポートより
        OUT     POUTPT,AL

        RET                             おわり
```

outwpルーチン

GDCには，ワード長のデータをパラメータとして，送出することが多いため，ワード出力用のルーチンを作りました。

```
OUTWP:
        MOV     AX,WRDPAR             } 上位バイトを送出
        MOV     PARMTR,AL
        CALL    OUTP
        MOV     PARMTR,AH             } 下位バイトを送出
        CALL    OUTP

        RET                             おわり
```

以上が，ラインルーチンです。これをテストするルーチンを以下に示します。このテストルーチンを動かせば，いかにLine文が速いか，認識できるでしょう。

```
TEST:
        XOR     AX,AX
        MOV     DS,AX
        MOV     COMMAND,0DH           } Startコマンド
        CALL    OUTC

        XOR     AX,AX
MAIN:   MOV     X1,AX
        MOV     BX,639                } x2 = 639 - ax
        SUB     BX,AX
        MOV     X2,BX
        XOR     BX,BX
        MOV     Y1,BX                   y1 = 0
        MOV     BX,199
        MOV     Y2,BX                   y2 = 199

        PUSH    AX
        AND     AL,3                  } color = al and 3
        MOV     COLOR,AL
        CALL    LINE                    lineルーチンをコール
        POP     AX
        INC     AX                      ax = ax + 1
        CMP     AX,640                } ax ≒ 640ならばmainへ
        JNE     MAIN
                                        axをクリア
        XOR     AX,AX
MAIN2:  MOV     BX,639                  x1 = 639
        MOV     X1,BX
        MOV     Y1,AX                   y1 = ax
        XOR     BX,BX
        MOV     X2,BX                   x2 = 0
        MOV     BX,199
        SUB     BX,AX                 } y2 = 199 - ax
        MOV     Y2,BX
        PUSH    AX
```

```
AND     AL,3          } color = al and 3
MOV     COLOR,AL      }
CALL    LINE            lineルーチンをコール
POP     AX              ax = ax + 1
INC     AX            } ax ≒ 200ならばmain 2へ
CMP     AX,200        }
JNE     MAIN2           おわり
```

直線を，色に変えながら画面全体に描くプログラムです。BASICでは，次のようになるでしょう。

1000 for x=0 to 639:

line (x, 0)−(639−x, 399), 2 ^(x mod 3):

next

1010 for y=0 to 199:

line (639, y)−(0, 199−y), 2 ^ (y mod 3):

next

BASICからcallする場合，プログラムの最後は，retではなくiretにしなくてはなりません。

CP/M86のユーザは，このリストを入れて，

ASM86　ファイル名 ⏎

GENCMD　ファイル名 ⏎

ファイル名 ⏎

とすれば，実行できます。CP/M86をもっていない人のために，アセンブラのリスティングファイルを載せておきます（リスト4-2）。

このプログラムは説明用に作ってあるので，BASICのプログラムのロジックをそのままとり入れました。mul命令を使ったところは，シフトに置き換えたり，テーブルサーチで高速化したりすることが可能です。そのほかパラメータをレジスタで渡すようにしたり，サブルーチン群を縦に展開したりすれば，少しはスピードが上がるでしょう。このままでも，十分に高速なのはご覧になったとおりです。

リスト4-2

```
;
; GDC DEMONSTRATION
;

STATPT  EQU     0A0H    ; STATUS READ PORT=A0H
COUTPT  EQU     0A2H    ; COMMAND OUT PORT=A2H
POUTPT  EQU     0A0H    ; PARAMETER OUT PORT=A0H

;
; CODE SEGMENT FOR LINE ROUTINE
;
        CSEG    0
        ORG     0A000H




TEST:
        XOR     AX,AX
        MOV     DS,AX
        MOV     COMMAND,0DH     ; START COMMAND
        CALL    OUTC

        XOR     AX,AX
MAIN:   MOV     X1,AX
        MOV     BX,639
        SUB     BX,AX
        MOV     X2,BX
        XOR     BX,BX
        MOV     Y1,BX
        MOV     BX,199
        MOV     Y2,BX

        PUSH    AX
        AND     AL,3
        MOV     COLOR,AL
        CALL    LINE
        POP     AX
        INC     AX
        CMP     AX,640
        JNE     MAIN

        XOR     AX,AX
MAIN2:  MOV     BX,639
        MOV     X1,BX
        MOV     Y1,AX
        XOR     BX,BX
        MOV     X2,BX
        MOV     BX,199
        SUB     BX,AX
        MOV     Y2,BX
        PUSH    AX

        AND     AL,3
        MOV     COLOR,AL
        CALL    LINE
        POP     AX
        INC     AX
        CMP     AX,200
        JNE     MAIN2
```

```
;
; LINE ROUTINE LINE(X1,Y1)-(X2,Y2),COLOR
;
; COLOR=0 BLUE COLOR=1 RED COLOR=2 GREEN
;
LINE:
        CALL    TEXTW
        CALL    WRITE
        CALL    INIT
        CALL    CULADR
        CALL    CSRW
        CALL    CULDIR
        CALL    CULPAR
        CALL    VECTW
        CALL    VECTE
        RET

TEXTW:  MOV     COMMAND,78H
        CALL    OUTC
        MOV     PARMTR,0FFH
        CALL    OUTP
        CALL    OUTP
        RET

WRITE:
        MOV     COMMAND,20H
        CALL    OUTC
        RET

INIT:

        MOV     AX,X2
        CMP     AX,X1
        JGE     INIT1

        XCHG    X1,AX
        MOV     X2,AX
        MOV     AX,Y1
        XCHG    Y2,AX
        MOV     Y1,AX
INIT1:  MOV     AX,X1
        CMP     AX,X2
        JNE     INIT2

        MOV     AX,Y2

        CMP     AX,Y1
        JGE     INIT3

        XCHG    Y1,AX
        MOV     Y2,AX

INIT2:
INIT3:
        RET

CULADR:
        MOV     AL,COLOR
        CMP     AL,0
        JNE     CUL1

        MOV     BX,4000H

        JMPS    CUL2

CUL1:   CMP     AL,1
        JNE     CUL3
```

```
        MOV     BX,8000H
        JMPS    CUL2
CUL3:
        MOV     BX,0C000H
CUL2:   MOV     EAD,BX
        MOV     AX,X1
        MOV     CL,4
        SHR     AX,CL

        ADD     EAD,AX

        MOV     AX,40
        MOV     BX,Y1
        MUL     BX

        ADD     EAD,AX

        MOV     AX,X1
        AND     AL,0FH
        MOV     DAD,AL
        RET

CSRW:
        MOV     COMMAND,49H

        CALL    OUTC

        MOV     AX,EAD
        MOV     WRDPAR,AX
        CALL    OUTWP

        MOV     AL,DAD
        MOV     CL,4
        SHL     AL,CL

        MOV     PARMTR,AL
        CALL    OUTP
        RET



CULDIR:
        MOV     AX,X2
        SUB     AX,X1
        MOV     DELTA_X,AX


        MOV     AX,Y2
        SUB     AX,Y1
        MOV     DELTA_Y,AX

        CMP     AX,0
        JLE     DIR1

        MOV     BX,DELTA_X
        CMP     AX,BX
        JLE     DIR2

        MOV     DIR,0

        MOV     AX,DELTA_X
        XCHG    DELTA_Y,AX
        MOV     DELTA_X,AX

        JMPS    DIR3
```

```
  :
  :
DIR2:   MOV     DIR,1
        JMPS    DIR3

DIR1:   NEG     DELTA_Y
        MOV     BX,DELTA_Y

        CMP     BX,DELTA_X
        JG      DIR4

        MOV     DIR,2
        JMPS    DIR3

DIR4:   MOV     DIR,3

        MOV     AX,DELTA_X
        XCHG    DELTA_Y,AX
        MOV     DELTA_X,AX

DIR3:
        RET

CULPAR:
        MOV     AX,DELTA_X
        MOV     DC_REG,AX

        MOV     AX,DELTA_Y
        SAL     AX,1
        SUB     AX,DELTA_X
        MOV     D_REG,AX

        MOV     AX,DELTA_Y
        SUB     AX,DELTA_X
        SAL     AX,1
        MOV     D2_REG,AX

        MOV     AX,DELTA_Y
        SAL     AX,1
        MOV     D1_REG,AX
        RET
VECTW:
        MOV     COMMAND,4CH

        CALL    OUTC

        MOV     AL,DIR
        OR      AL,8

        MOV     PARMTR,AL
        CALL    OUTP

        MOV     AX,DC_REG
        MOV     WRDPAR,AX
        CALL    OUTWP

        MOV     AX,D_REG
        MOV     WRDPAR,AX
        CALL    OUTWP

        MOV     AX,D2_REG
        MOV     WRDPAR,AX
        CALL    OUTWP

        MOV     AX,D1_REG
```

```
        MOV     WRDPAR,AX
        CALL    OUTWP
        RET

VECTE:
        MOV     COMMAND,6CH

        CALL    OUTC

        RET

;
; END OF LINE ROUTINE
;

OUTC:

OUTCLP: IN      AL,STATPT
        AND     AL,2
        JNZ     OUTCLP

        MOV     AL,COMMAND
        OUT     COUTPT,AL

        RET

OUTP:

OUTPLP: IN      AL,STATPT
        AND     AL,2
        JNZ     OUTPLP

        MOV     AL,PARMTR
        OUT     POUTPT,AL

        RET

OUTWP:
        MOV     AX,WRDPAR
        MOV     PARMTR,AL
        CALL    OUTP
        MOV     PARMTR,AH
        CALL    OUTP

        RET

 ;
 ; DATA SEGMENT FOR LINE ROUTINE
 ;



         DSEG    0
         ORG     0B000H
 COMMAND DB      0
 PARMTR  DB      0
 X1      DW      0
 Y1      DW      0
 X2      DW      0
 Y2      DW      0

 COLOR   DB      0

 EAD     DW      0
 DAD     DB      0
```

```
DELTA_X DW       0
DELTA_Y DW       0

DIR     DB       0

DC_REG  DW       0
D_REG   DW       0
D2_REG  DW       0
D1_REG  DW       0

WRDPAR  DW       0

;
;



        END
```

2 Circleのプログラム

前出のLineルーチンはいかがでしたか。いかにPC-9801のグラフィックが，高速であるか分かったと思います。ここではGDCを使った高速Circleルーチンを，

- コマンド，パラメータの設定法
- BASICによるプログラム例
- アセンブラによるプログラム例

に分けて説明します。例によってBASICのプログラムを8086のアセンブラに落としていくわけです。ゲームなどで高速にサークルを描く必要があることはよくあるでしょうが，そんなときに大いに利用してください。

Circleについて

円を描くプログラムは，

$$y=\sqrt{r^2-x^2}$$

を，$-r \leqq x \leqq r$の区間でプロットする方法や

$$x=r\ \cos\theta,\ y=r\ \sin\theta$$

(円のパラメトリックな表現)

を$0 \leqq \theta \leqq 2\pi$の区間でプロットする方法，あるいはDDAによる方法などいろいろありますが，多くのアルゴリズムでは実数演算や，三角関数が必要であり，高速に描くには不向きなものが多いのです（実数演算は高速なプログラムあるいはハードウェアでの実現に対し，致命的です)。

そんな中でIBMのBresenhamの開発したアルゴリズムはわずかなadd, subtract, shiftでCircleを描くことができ，ハードロジックでcircle generatorを組んだり，アセンブラでプログラムしたりする場合に，よく使われています。私の知る限りでは，このBresenhamのアルゴリズムが，最も速いアルゴリズムのようです。

もともとこのアルゴリズムはプロッタ制御用に開発されたものですが，ラスターグラフィックに応用できることはいうまでもありません。

7220Dのサークル機能は，どうもこのBresenhamのアルゴリズムを使って

いるようです。Bresenham はほかに Line を高速に描画するアルゴリズムも発表していますが，GDC の Line 発生のロジックもこの Bresenham のアルゴリズムを採用しているようです。

これは GDC の DC，D，D2，D1，DM レジスタへの設定値を見るとよく分かります。Bresenham のアルゴリズムの初期設定値と酷似しています。Bresenham のアルゴリズムは別の機会に譲るとして，ここでは参考文献を紹介するにとどめます。

〈Bresenham, J. E., "Algorithm for Computer Control Program of Digital Plotter," IBM Syst. J., 4(1)1965. pp. 25—30.
Bresenham, J. E., "A Linear Algorithm for Incremental Digital Display of Circular Arcs," Communications of the ACM, 20(2), February 1977, pp. 100—106〉

Circleコマンド

PC-9801 にもサークル命令はあります。しかし，GDC のサークル機能を使って描画しているわけではありません。これは例によって「ビューポート内しか描画しない」という制約があるためで，結局 1 ドットずつ点をプロットしているようです。また，ディスプレイのアスペクト比（縦と横のドット比）が，1 対 1 でない場合（たとえば，640×200設定時），GDC でサークルを描くと，円ではなく，楕円になることも GDC でサークルを描かせなかった理由の 1 つでしょう。

しかし，そのためサークル描画はきわめて遅いものとなっています。これでは，シミュレーション，アニメーション，ゲームなどでは使いづらいでしょう。

同じ GDC を使った，EPSON の QC-10 のサークルはスクリーンから円がはみ出すかどうかをまず判断して，円がスクリーンの中に完全に入っているときは GDC のサークル命令で描きます。また，少しでもはみ出す場合は交点を計算して描画するといった二段構えのプログラムになっており，はみ出さない場合は，きわめて高速に，サークルを描画します。一度試してください（このほか QC-10 にはズームリード機能がついているなど，なかなかおもしろい機械です。PC-9801 では，ズームライトは可能ですが，ズームリードはできない

ようです)。

BASIC のコマンドではサポートされてないにしても，GDC を直接ドライブしてやれば，サークルは高速に描画できます。そこで，その方法を解説しましょう。

1.コマンド，パラメータの設定法

GDC では一度に描画できるのは0°～45°の領域に限られます。そのため，円を描画するのに，8分の1ずつ8回にわけて描画します。円を描画するには GDC に次のコマンドを送ります。

TEXTWコマンド　円の線の種類を指定します。パラメータで FFFFH を送れば実線，3333H を送れば破線となります（BASIC の Line 文の線種と同様)。

WRITEコマンド　円を描画するのか，消すのか，XOR をとって描画するのか，などを指定するコマンドです。ここでは OR をとって描画するため，20H を使います（22H ならば消去)。

CSRW（カーソルライト）コマンド　1/8円をどこから描画するかを指定するコマンドです。パラメータとしては，描画を開始するドットの GDC アドレス，ドットアドレスを送出します。298ページで述べたように GDC から見た VRAM アドレスは CPU から見た VRAM のアドレスとは全然違うので注意が必要です（GDC から見た場合，blue 4000H～red 8000H～ green C000H～ なお，1アドレス＝16bitです。右が MSB！)。

VECTWコマンド　描画の種類（円か，直線か，四辺形かなど)，描画方向（0～7の8種類）描画用パラメータレジスタ DC, D, D2, D1, DM のセットを行います。各レジスタの設定値は，円描画の場合，半径を r とすれば，

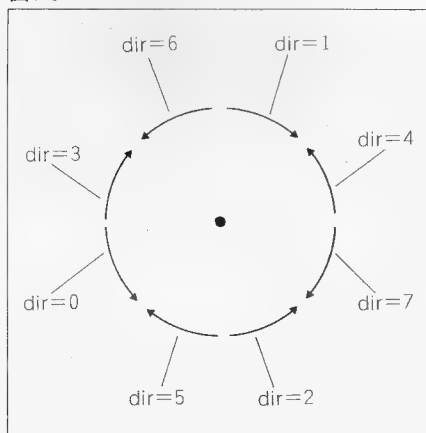
$DC=r/\sqrt{2}\uparrow$（↑は切り上げ）

$D=r-1$

$D2=2(r-1)$

$D1=-1$

$DM=0$

です。$DC=r/\sqrt{2}\uparrow$，$DM=0$ は，$DC=r\ \sin\ (45°)$，$DM=r\ \sin\ (0°)$ を意味し，0°～45°の円弧を描画することになります。(すなわち1/8円) 45°，0°を

ほかに変更することで，任意の円弧を描けます（もちろん0°～45°の間)。

VECTEコマンド　GDC に描画開始を指令するコマンドです。

以上のコマンド，パラメータを設定しつつ，8回描画すれば，円は描けるわけですが，毎回設定するのでは時間がかかります。8つの GDC の描画方向は図4－5のようになっていますが，7と4，1と6，3と0，5と2は，同じ描画開始アドレス，ドットアドレスをもっているため，ひとまめにして考えます。また TEXTW，WRITE コマンドは最初に一度セットすればよい（1/8円を描画するごとに呼ぶ必要はない）ので，プログラムの先頭に置きます。また，VECTW コマンド用パラメータも，描画方向以外は変更がないので，やはり最初に計算しておきます。

以上のことから，プログラムのフローチャートは図4－6のようになります。

2.BASICによるプログラム例

いまのフローチャートを，そのまま BASIC で書くと図4-7のようになりま

BASICで組んだ例

```
110 FOR R=1 TO 199:CIRCLE(R*2,200),R,2^(R MOD 3):NEXT R
```

図4-6

す。このプログラムでは円の中心，半径を入力するようになっています（描画プレーンは，青を指定)。楕円でなく，真円にするには，640×400のモードでなければならないことに注意してください。このプログラムをRUNして，たとえば，320，200，150⏎と入力すれば，中心（320，200)，半径150の円が青で描画されます。プログラムはフローチャートのとおり，トップダウンに書いてあるので説明の必要はないでしょう。ここでもう一度GDCへコマンド，パラメータを渡すルーチンを説明しておきます。

＊OUT．COMMAND　これはGDCへコマンドを送出するルーチンで，変数COMMANDに入っているGDCコマンドを，GDCのFIFOがFullでないのを確認して送出するルーチンです。

＊OUT．PARAMETER　GDCへパラメータを送出するルーチンです。引数は変数PARAMETER。＊OUT．COMMANDルーチン同様，FIFO not fullを確認しています（もっともBASICで送る場合，FIFOがfullになること

図4-7 BASICによるプログラム

```
1000 *CIRCLE.ROUTINE
1020   GOSUB *INPUT.PARAMETER
1030   GOSUB *TEXTW.ROUTINE
1040   GOSUB *WRITE.ROUTINE
1050   GOSUB *SET.VECTW.DATA
1060   GOSUB *DRAW.ARC.DIR.7.4
1070   GOSUB *DRAW.ARC.DIR.1.6
1080   GOSUB *DRAW.ARC.DIR.3.0
1090   GOSUB *DRAW.ARC.DIR.5.2
1100 END
1110 '
1120 *INPUT.PARAMETER INPUT "CIRCLE(X.Y).R ";X.C,Y.C,RADIUS:RETURN
1130 '
1140 *TEXTW.ROUTINE
1150   COMMAND=&H78:GOSUB *OUT.COMMAND
1160   PARAMETER.W=&HFFFF:GOSUB *OUT.WORD.PARAMETER
1170 RETURN
1172 '
1174 *WRITE.ROUTINE COMMAND=&H20:GOSUB *OUT.COMMAND:RETURN
1175 '
1180 *SET.VECTW.DATA
1190  DC.REG=RADIUS/SQR(2)+1 : D.REG=RADIUS-1
1210 D2.REG=2*D.REG : D1.REG=-1:DM.REG=0
1240 RETURN
1250 '
1260 *DRAW.ARC.DIR.7.4
1270   XCRD=X.C+RADIUS : YCRD=Y.C : GOSUB *CUL.CSRW.PARAMETER
1280   GOSUB *CSRW.ROUTINE
1290   DIRECTION=7 : GOSUB *VECTW.ROUTINE
1292   GOSUB *VECTE.ROUTINE
1295   GOSUB *CSRW.ROUTINE
1300   DIRECTION=4 : GOSUB *VECTW.ROUTINE
1302   GOSUB *VECTE.ROUTINE
1310 RETURN
1320 '
2260 *DRAW.ARC.DIR.1.6
2270   XCRD=X.C : YCRD=Y.C-RADIUS : GOSUB *CUL.CSRW.PARAMETER
2280   GOSUB *CSRW.ROUTINE
2290   DIRECTION=1 : GOSUB *VECTW.ROUTINE
2292   GOSUB *VECTE.ROUTINE
2295   GOSUB *CSRW.ROUTINE
2300   DIRECTION=6 : GOSUB *VECTW.ROUTINE
2302   GOSUB *VECTE.ROUTINE
2310 RETURN
2320 '
3260 *DRAW.ARC.DIR.3.0
3270   XCRD=X.C-RADIUS : YCRD=Y.C : GOSUB *CUL.CSRW.PARAMETER
3280   GOSUB *CSRW.ROUTINE
3290   DIRECTION=3 : GOSUB *VECTW.ROUTINE
3292   GOSUB *VECTE.ROUTINE
3295   GOSUB *CSRW.ROUTINE
3300   DIRECTION=0 : GOSUB *VECTW.ROUTINE
3302   GOSUB *VECTE.ROUTINE
3310 RETURN
3320 '
4260 *DRAW.ARC.DIR.5.2
4270   XCRD=X.C : YCRD=Y.C+RADIUS : GOSUB *CUL.CSRW.PARAMETER
4280   GOSUB *CSRW.ROUTINE
4290   DIRECTION=5 : GOSUB *VECTW.ROUTINE
4292   GOSUB *VECTE.ROUTINE
4295   GOSUB *CSRW.ROUTINE
4300   DIRECTION=2 : GOSUB *VECTW.ROUTINE
4302   GOSUB *VECTE.ROUTINE
4310 RETURN
4320 '
```

```
5000  *CUL.CSRW.PARAMETER
5010   EAD=&H4000+40*YCRD+XCRD ¥ 16 : DAD=XCRD MOD 16
5020 RETURN
5030 '
5040 *CSRW.ROUTINE
5050   COMMAND=&H49:GOSUB *OUT.COMMAND
5060   PARAMETER.W=EAD : GOSUB *OUT.WORD.PARAMETER
5080   PARAMETER=DAD*16 : GOSUB *OUT.PARAMETER
5090 RETURN
5100 '
6000 *VECTW.ROUTINE
6005   COMMAND=&H4C :GOSUB *OUT.COMMAND
6010   PARAMETER=&H20+DIRECTION : GOSUB *OUT.PARAMETER
6020   PARAMETER.W=DC.REG : GOSUB *OUT.WORD.PARAMETER
6030 '
6050   PARAMETER.W=D.REG : GOSUB *OUT.WORD.PARAMETER
6060   PARAMETER.W=D2.REG : GOSUB *OUT.WORD.PARAMETER
6070   PARAMETER.W=D1.REG : GOSUB *OUT.WORD.PARAMETER
6080   PARAMETER.W=DM.REG : GOSUB *OUT.WORD.PARAMETER
6090 RETURN
7000 '
7010 *VECTE.ROUTINE COMMAND=&H6C : GOSUB *OUT.COMMAND : RETURN
10000 *OUT.COMMAND WHILE INP(&HA0) AND 2:WEND:OUT &HA2,COMMAND:RETURN
10010 *OUT.PARAMETER  WHILE INP(&HA0) AND 2:WEND:OUT &HA0,PARAMETER:RETURN
20000 *OUT.WORD.PARAMETER
20010 PARAMETER=VAL("&H"+RIGHT$("000"+HEX$(PARAMETER.W),2))
20020  GOSUB *OUT.PARAMETER
20025  PARAMETER=VAL("&H"+LEFT$(RIGHT$("000"+HEX$(PARAMETER.W),4),2))
20027  GOSUB *OUT.PARAMETER
20030 RETURN
```

は，まずありません)。

＊OUT. WORD. PARAMETER　　GDCにはワード長のパラメータが多いので，それをパラメータ出力するルーチンです。引数は変数PARAMETER.W。

3.アセンブラによるプログラム例

BASICでのプログラム例を示しましたが，これは，単にロジックをチェックする意味しかありません。このアルゴリズムをアセンブラなり，コンパイラなどで書いて初めて意味をもちます。そこで，このサークルルーチンを，8086のアセンブラで記述してみます（リスト4-3)。例によって，BASICをそのまま落としていくので，最適化は，各自試みてください。ただ，このままでも，十分高速です。

プログラムはCP/M-86のアセンブラを使って開発しました。CP/M-86のアセンブラでは，コードセグメント，データセグメントを具体的に，設定しなくてもよいのです（OSが勝手に割り当ててくれます)。今回は，CP/M-86を使

ダンプリスト


```
A000 33 C0 8E D8 B4 40 CD 18 B4 42 B5 C0 CD 18 C7 06
A010 02 B0 C8 00 B8 01 00 C6 06 0C B0 00 A3 04 B0 8B
A020 D8 D1 E3 89 1E 00 B0 FE 06 0C B0 8A 0E 0C B0 80
A030 F9 03 75 05 C6 06 0C B0 00 50 E8 08 00 58 40 3D
A040 C7 00 75 D8 CF E8 13 00 E8 22 00 E8 28 00 E8 4F
A050 00 E8 7C 00 E8 A9 00 E8 D6 00 C3 C6 06 0A B0 78
A060 E8 A3 01 C7 06 1B B0 FF FF E8 B2 01 C3 C6 06 0A
A070 B0 20 E8 91 01 C3 A1 04 B0 BB 30 75 F7 E3 BB BA
A080 A5 F7 F3 40 A3 11 B0 A1 04 B0 48 A3 13 B0 D1 E0
A090 A3 15 B0 C7 06 17 B0 FF FF C7 06 19 B0 00 00 C3
A0A0 A1 00 B0 03 06 04 B0 A3 06 B0 A1 02 B0 A3 08 B0
A0B0 E8 AD 00 E8 E6 00 C6 06 10 B0 07 E8 FD 00 E8 3C
A0C0 01 E8 D8 00 C6 06 10 B0 04 E8 EF 00 E8 2E 01 C3
A0D0 A1 00 B0 A3 06 B0 A1 02 B0 2B 06 04 B0 A3 08 B0
A0E0 E8 7D 00 E8 B6 00 C6 06 10 B0 01 E8 CD 00 E8 0C
A0F0 01 E8 A8 00 C6 06 10 B0 06 E8 BF 00 E8 FE 00 C3
A100 A1 00 B0 2B 06 04 B0 A3 06 B0 A1 02 B0 A3 08 B0
A110 E8 4D 00 E8 86 00 C6 06 10 B0 03 E8 9D 00 E8 DC
A120 00 E8 78 00 C6 06 10 B0 00 E8 8F 00 E8 CE 00 C3
A130 A1 00 B0 A3 06 B0 A1 02 B0 03 06 04 B0 A3 08 B0
A140 E8 1D 00 E8 56 00 C6 06 10 B0 05 E8 6D 00 E8 AC
A150 00 E8 48 00 C6 06 10 B0 02 E8 5F 00 E8 9E 00 C3
A160 80 3E 0C B0 00 75 05 BB 00 40 EB 0F 80 3E 0C B0
A170 01 75 05 BB 00 80 EB 03 BB 00 C0 B8 28 00 8B 0E
A180 08 B0 F7 E1 03 C3 8B 1E 06 B0 B1 04 D3 FB 03 C3
A190 A3 0D B0 A1 06 B0 24 0F A2 0F B0 C3 C6 06 0A B0
A1A0 49 E8 62 00 A1 0D B0 A3 1B B0 E8 71 00 A0 0F B0
A1B0 B1 04 D2 E0 A2 0B B0 E8 58 00 C3 C6 06 0A B0 4C
A1C0 E8 43 00 B0 20 02 06 10 B0 A2 0B B0 E8 43 00 A1
A1D0 11 B0 A3 1B B0 E8 46 00 A1 13 B0 A3 1B B0 E8 3D
A1E0 00 A1 15 B0 A3 1B B0 E8 34 00 A1 17 B0 A3 1B B0
A1F0 E8 2B 00 A1 19 B0 A3 1B B0 E8 22 00 C3 C6 06 0A
A200 B0 6C E8 01 00 C3 E4 A0 24 02 75 FA A0 0A B0 E6
A210 A2 C3 E4 A0 24 02 75 FA A0 0B B0 E6 A0 C3 A1 1B
A220 B0 A2 0B B0 E8 E8 FF 88 26 0B B0 E8 E4 FF C3
```

mon ↵
]C0 ↵
したあとで打ちこんで下さい。
実行方法は，BASICにもどったあと
DEF SEG=0 ↵
a = & HA000 ↵
call a ↵

っていない人のために，プログラム中で指定しています。

アセンブラを持っている人はいまのテキストを入れればよいのですが，もってない人のために，ダンプリストもつけておきます。

スピードを比較するために，BASICで，いまのアセンブラによるプログラムと同じようなルーチンを載せておきます（図4－7）。実行速度は，

BASIC　1分18秒

アセンブラ　1秒ちょっと

となりました。実際に動かしていかに速いか比べてみてください。

リスト4-3　アセンブラによるプログラムリスト(CP/M-86)

```
;
; CIRCLE DEMO
;


; CONSTANTS FOR CIRCLE ROUTINE

;
COUTPT  EQU     0A2H        a2HはGDCのコマンド出力ポート
POUTPT  EQU     0A0H        a0HはGDCのパラメータ出力ポート
STATPT  EQU     0A0H        a0HはGDCのステータスリードポート

        CSEG    0           コードセグメントを0000Hにわりあてる
        ORG     0A000H      アセンブラのコード生成開始アドレス = a000H

                            サークルルーチンを利用したデモルーチン
TEST:
;
;
; TEST PROGRAM
;
        XOR     AX,AX       ax = 0
        MOV     DS,AX       データセグメントを0にセット

        MOV     AH,40H  }   グラフィックの表示開始
        INT     18H     }   (CP/M-86の初期状態では,グラフィック画面は
                            マスクされているので,これが必要)

        MOV     AH,42H  }   グラフィックスクリーンをカラー640×400にセット
        MOV     CH,0C0H }   (CP/M-86の初期状態では,カラー640×200になっている)
        INT     18H     }

        MOV     Y_C,200         y.c = 200
        MOV     AX,1            ax = 1
        MOV     COLOR,0         color = 0

TESTL:  MOV     RADIUS,AX       radius = ax
        MOV     BX,AX       }   bx = 2 * radius
        SAL     BX,1        }
        MOV     X_C,BX          x.c = 2 * radius
        INC     COLOR           color = color + 1
        MOV     CL,COLOR    }
        CMP     CL,3        }   if color ≠ 3  then test 1
        JNE     TEST1       }

        MOV     COLOR,0         color = 0

TEST1:
        PUSH    AX              axを保存
        CALL    CIRCLE          サークルルーチンをコール
        POP     AX              axを復帰させる

        INC     AX              ax = ax + 1
        CMP     AX,199          ax = 199  ?

        JNE     TESTL           if ax ≠ 199 then test l

        IRET                    終わり(CP/M-86あるいはBASICから呼ぶ場合プログラムの終わ
;                                    りはretではなくiretを必ず使う)
; END OF TEST ROUTINE
;

CIRCLE:
```

```
;
; CIRCLE MAIN ROUTINE
;
; ARGUMENTS X_C,YC,RADIUS,COLOR
;
    入力：中心座標 ,   半径,     色

; ,WHERE COLOR --> 0:BLUE 1:RED 2:GREEN
;
        CALL     TEXTW            線の種類を決めるルーチン(破線, 実線等)
        CALL     WRITE            描画か消去を決めるルーチン
        CALL     SET_VECTW_DATA   VECTWコマンド用ハラメータの計算
        CALL     ARC74            描画方向7, 4の1/8円弧の描画
        CALL     ARC16              〃   1, 6      〃
        CALL     ARC30              〃   3, 0      〃
        CALL     ARC52              〃   5, 2      〃

        RET      終わり
;
;       END OF MAIN ROUTINE
;

                 TEXTWルーチン
TEXTW:           線の種類(実線, 破線, 一点鎖線等)を決めるルーチン
;
; TEXTW ROUTINE
;
; SEND TEXTW COMMAND AND PARAMETERS TO GDC
;
                           CP M, CP M-86のアセンブラでは！で一行に何行でも書ける
        MOV      COMMAND,78H    (!)CALL  OUTC      78HはGDCのTEXTWコマンド

        MOV      WRDPAR,0FFFFH  ! CALL  OUTWP
                 ワード長のパラメータをWRDPARに代入し,outwp(ワード長パラメータ出力)
        RET  →   終わり

;
; END OF TEXTW ROUTINE
;
                 WRITEルーチン
WRITE:           描画か消去を指定するルーチン
;
; WRITE ROUTINE
;
; SEND WRITE COMMAND TO GDC

        MOV      COMMAND,20H     ! CALL  OUTC     ! RET
                         20HはGDCのWRITEコマンド
;
; END OF WRITE ROUTINE
;


SET_VECTW_DATA:
;
;       SET DC.reg D.reg DI.reg DM.reg data
;                        描画パラメータラジスタ：DC, D, D2, DI, DM
        MOV      AX,RADIUS      ax = radius
        MOV      BX,10000*3     bx = 10000 * 3
        MUL      BX             dx : ax = 10000 * 3 * radius    DC = 半径 √2を計算する
        MOV      BX,14142*3     bx = 10000 * 3 * √2             ために
        DIV      BX             ax = radius √2                  半径30000 (3 ・14142)
        INC      AX             ax = ax + 1                     としている
        MOV      DC_REG,AX      DC REG = radius √2

        MOV      AX,RADIUS   }
        DEC      AX          }  D REG = 半径 − 1
        MOV      D_REG,AX    }
```

```
        SAL     AX,1          } D2  REG = 2・(半径－1)
        MOV     D2_REG,AX

        MOV     D1_REG,-1       D1  REG = －1
        MOV     DM_REG,0        DM  REG = 0

        RET     終わり
;



ARC74:          描画方向7, 4の1/8円弧を描くルーチン
        MOV     AX,X_C          ax = x  c
        ADD     AX,RADIUS       ax = 半径 + x  c
        MOV     X_CRD,AX        x  CRD = 半径 + x  c

        MOV     AX,Y_C        } y  CRD = y  c
        MOV     Y_CRD,AX

        CALL    CUL_CSRW_PAR  CSRW用パラメータの計算(EAD, DADをセット)
        CALL    CSRW          CSRWコマンドをGDCに送出
        MOV     DIR,7         DIR = 7
        CALL    VECTW         VECTWコマンド & パラメータをGDCに送出
        CALL    VECTE         VECTEコマンドをGDCに送出

        CALL    CSRW          CSRWコマンド, パラメータの送出
        MOV     DIR,4         DIR = 4
        CALL    VECTW         VECTWコマンド, パラメータの送出
        CALL    VECTE         VECTEコマンドの送出

        RET     終わり



ARC16:          描画方向1, 6の1/8円弧を描画
        MOV     AX,X_C        } x  CRD = x  c
        MOV     X_CRD,AX

        MOV     AX,Y_C
        SUB     AX,RADIUS     } y  CRD = y  c－半径
        MOV     Y_CRD,AX

        CALL    CUL_CSRW_PAR  CSRW用パラメータ(EAD, DAD)の計算
        CALL    CSRW          CSRWコマンド, パラメータの送出
        MOV     DIR,1         DIR = 1
        CALL    VECTW         VECTWコマンド, パラメータの送出
        CALL    VECTE         VECTEコマンド(描画開始)

        CALL    CSRW          CSRWコマンド, パラメータの送出
        MOV     DIR,6         DIR = 6
        CALL    VECTW         VECTWコマンド, パラメータの送出
        CALL    VECTE         VECTEコマンド発行(描画開始)

        RET     終わり
```

```
ARC30:          描画方向3, 0の1/8円弧を描画
        MOV     AX,X_C         }
        SUB     AX,RADIUS      } x CRD = x c − 半径
        MOV     X_CRD,AX       }

        MOV     AX,Y_C         }
        MOV     Y_CRD,AX       } y CRD = y c

        CALL    CUL_CSRW_PAR  CSRW用パラメータ(EAD, DAD)の計算
        CALL    CSRW          CSRWコマンド, パラメータの送出
        MOV     DIR,3         DIR = 3
        CALL    VECTW         VECTWコマンド, パラメータ送出
        CALL    VECTE         VECTEコマンド送出(描画開始)

        CALL    CSRW          CSRWコマンド, パラメータの送出
        MOV     DIR,0         DIR = 0
        CALL    VECTW         VECTWコマンド, パラメータの送出
        CALL    VECTE         VECTEコマンド発行(描画開始)

        RET     終わり



ARC52:          描画方向5, 2の1/8円弧の描画
        MOV     AX,X_C         }
        MOV     X_CRD,AX       } x CRD = x c

        MOV     AX,Y_C         }
        ADD     AX,RADIUS      } y CRD = y c + 半径
        MOV     Y_CRD,AX       }

        CALL    CUL_CSRW_PAR  CSRW用パラメータ(EAD, DAD)の計算
        CALL    CSRW          CSRWコマンド発行
        MOV     DIR,5         DIR = 5
        CALL    VECTW         VECTWコマンド, パラメータ送出
        CALL    VECTE         VECTEコマンド発行(描画開始)

        CALL    CSRW          CSRWコマンド, パラメータ送出
        MOV     DIR,2         DIR = 2
        CALL    VECTW         VECTWコマンド, パラメータ送出
        CALL    VECTE         VECTEコマンド発行(描画開始)
        RET     終わり

CUL_CSRW_PAR:   CSRW用パラメータ(EAD, DAD)の計算をするルーチン
 :
 :
        CMP     COLOR,0       }
        JNE     CULC1         } if color < > 0 then culc 1へ

        MOV     BX,4000H      } bx = 4000H(4000Hは青のVRAMのSTARTアドレス)
        JMPS    CULC2         } としてculc 2へ(JMPSはセグメント内ジャンプ)

CULC1:  CMP     COLOR,1       }
        JNE     CULC3         } if color < > 1 then culc 3へ

        MOV     BX,8000H      } bx = 8000H (8000Hは赤のVRAMのSTARTアドレス)
        JMPS    CULC2         } としてculc 2へ

CULC3:  MOV     BX,0C000H     } color = 0, 1以外ならば,bx = C000H, (C000Hは緑のVRAMの
                              } STARTアドレス)
CULC2:  MOV     AX,40         }
```

```
        MOV     CX,Y_CRD        ax = 40 * y  CRD (これはshiftとADDで高速化できる)
        MUL     CX
        ADD     AX,BX           ax = V-RAM START ADDRESS + 40 * y  CRD
        MOV     BX,X_CRD
        MOV     CL,4            bx = x  CRD
        SAR     BX,CL

        ADD     AX,BX           EAD = V-RAM START ADDRESS + 40 * y  CRD
        MOV     EAD,AX                + x  CRD 16
        MOV     AX,X_CRD
        AND     AL,15           DAD = x  CRD mod 16
        MOV     DAD,AL

        RET     終わり




CSRW:                   CSRWコマンド，パラメータを送出するルーチン
;
        MOV     COMMAND,49H     49HはCSRWコマンド
        CALL    OUTC            49Hをコマンド出力

        MOV     AX,EAD          WRDPAR = EAD
        MOV     WRDPAR,AX
        CALL    OUTWP           EADをワードパラメータ出力

        MOV     AL,DAD
        MOV     CL,4
        SAL     AL,CL           DADを左に4回シフトしたものをパラメータ出力
        MOV     PARMTR,AL
        CALL    OUTP

        RET     終わり
VECTW:                  VECTWコマンド，パラメータを送出するルーチン
        MOV     COMMAND,4CH     4CHはVECTWコマンド
        CALL    OUTC            4CHをGDCにコマンド出力

        MOV     AL,20H
        ADD     AL,DIR          20h + dir をパラメータ出力
        MOV     PARMTR,AL
        CALL    OUTP

        MOV     AX,DC_REG
        MOV     WRDPAR,AX       DC REGをワード長のパラメータとして出力

        CALL    OUTWP

        MOV     AX,D_REG
        MOV     WRDPAR,AX       D REGをワード長のパラメータとして出力
        CALL    OUTWP

        MOV     AX,D2_REG
        MOV     WRDPAR,AX       D2 REGをワード長のパラメータとして出力
        CALL    OUTWP
```

```
        MOV     AX.DI_REG      } DI REGをワード長のハラメータとして出力
        MOV     WRDPAR,AX      }
        CALL    OUTWP          }

        MOV     AX,DM_REG      } DM REGをワード長のパラメータとして出力
        MOV     WRDPAR,AX      }
        CALL    OUTWP          }

        RET     終わり
```

```
VECTE:          VECTEコマンドをGDCへ送出するルーチン

        MOV     COMMAND,6CH  6CHはVECTEコマンド
        CALL    OUTC         6CHをコマンド送出
        RET     終わり

OUTC:           GDCへコマンドを出力するルーチン

OUTCLP: IN      AL,STATPT      } FIFOがfullでなくなるまでまつ
        AND     AL,2           } FIFO fullはA0Hのbit 1
        JNZ     OUTCLP         }

        MOV     AL,COMMAND     } A2Hにcommandをout
        OUT     COUTPT,AL      }

        RET     終わり

OUTP:
                GDCへパラメータを出力するルーチン
```

```
OUTPLP: IN      AL,STATPT      } FIFO not fullを待つ
        AND     AL,2           }
        JNZ     OUTPLP         }

        MOV     AL,PARMTR      } parmtrをA0Hに出力
        OUT     POUTPT,AL      }       パラメータ出力ポート
```

```
        RET


OUTWP:              ワード長のパラメータを出力するルーチン
        MOV     AX,WRDPAR  ┐
        MOV     PARMTR,AL  │ WRDPARの下位8bitを送出
        CALL    OUTP       ┘
        MOV     PARMTR,AH  ┐
        CALL    OUTP       ┘ WRDPARの上位8bitを送出

        RET


                データセグメント
        DSEG    0               データセグメントを0000Hに設定
        ORG     0B000H          ワークエリアをB000H～に設定
X_C     DW      0          ┐
Y_C     DW      0          ┘ 中心座標
RADIUS  DW      0               半径

X_CRD   DW      0          ┐
Y_CRD   DW      0          ┘ CSRW用の引数

COMMAND DB      0
PARMTR  DB      0

COLOR   DB      0

EAD     DW      0
DAD     DB      0

DIR     DB      0


DC_REG  DW      0          ┐
                           │
D_REG   DW      0          │ 描画パラメータ
                           │
D2_REG  DW      0          ┘

D1_REG  DW      0

DM_REG  DW      0

WRDPAR  DW      0


        END     終わり
```

さて，サークルルーチンはいかがでしたか。動かしてみると分かりますが，きわめて高速でGDC7220の実力をまざまざと思い知らされます。BASICのサークルは1つひとつ「じわーっ」と描画し，見るほうもいらいらしてきますが，アセンブラでGDCを直接ドライブすると，あるサークルのどこを描画中かなどは見えないし，連続して描画するとどのサークルを描画中かも分かりません。

それほどGDCのサークルは高速なのです。

GDCは円弧を描く機能ももっています。いや，むしろ円弧を描く機能を利用して円を描画しているといったほうが正しいでしょう。ただ，例によって，BASICのように開始アングルを指定して簡単に描画できるわけではありません。GDCに関するわれわれの印象は高機能，高速ですが，ソフトもハードも大変といった感じです。GDCを活用するつもりなら簡単にやってやろう，という安易な気持ちは捨てたほうがよいでしょう。GDCだけのスペックだけ見ると，こんなこともできる，あんなこともできる，と書いてありますが，ある機能を実現するためにどれだけの努力が必要かプログラムを組んでみて初めて思い知らされます。決して，GDCのスペックからそれを読みとることはできません。

たとえば，ズームリードやパニング（全方向スクロール）を実行するためにはTTLの外づけ回路が必要です。これも組んでみて初めて気がつくことです。

ただ，これはハードもソフトも0から出発した場合です。適切なハードができ上がり，基本的なソフトさえあればGDCは確かに高速・高機能です。PC-9801に関していえば適切なハードがあるのですから，あと残るのはソフトだけです。

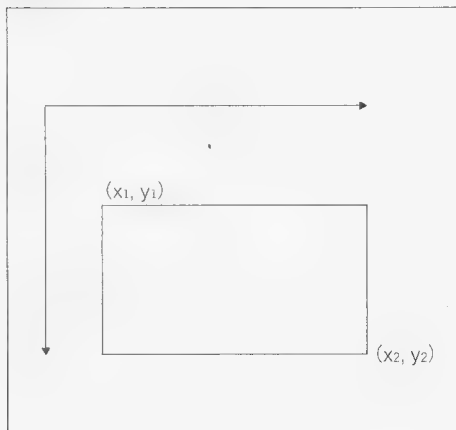
話を円弧にもどしましょう。円弧を描画させるのに必要なパラメータはr sin(T)です。これを計算するには遅くてよいので，8087にさせるか，精度はともかく高速性が必要ならテーブルサーチによって行うことです。あと開始アングル，終了アングルがどの描画方向にあるかなどを計算してパラメータを求め，描画ということになります。

BOXのプログラム

ここではBOXルーチンを紹介します。Lineルーチンを使って4回描画すれば確かにBOXになりますが，それでは4回描画パラメータをセットしなければならず，時間がかかります。GDCの機能に四辺形描画があるのでこれを利用します（さすがに多機能)。

四辺形描画は直線や円に比べると比較的簡単です。GDCのコマンド・パラメータの送り方の復習と考えてほしいのです。

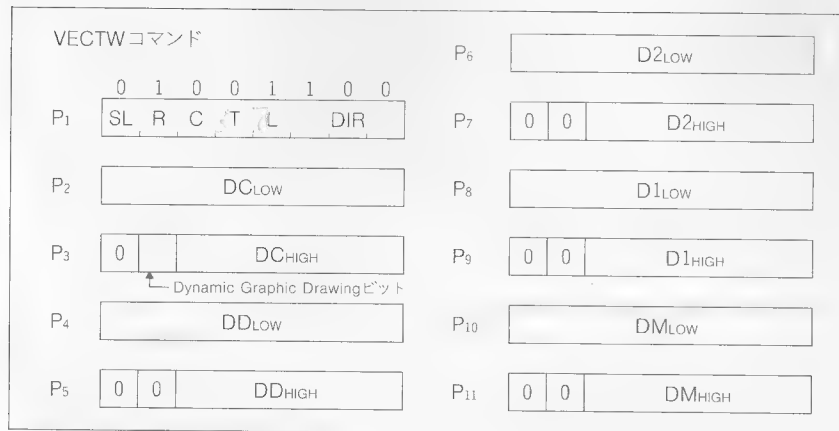
いま図4-8のような四辺形を書くことにします。

図4-8

BASICでいえば

Line$(x_1, y_1)-(x_2, y_2)$.,B

GDCのグラフィック描画で，描画図形（円弧か，Lineか BOXかなど）を決定するのはVECTWコマンドです。VECTWコマンドは厳密には描画方向，描画種類，描画パラメータレジスタの設定を行います。少しVECTWをつっこんで解説してみましょう。

VECTWコマンド(4ch)に続いて送られる8ビット長のパラメータは図4-9のようになっています。

図4-9

第1パラメータは,

SL	R	C	T	L	←DIR→

ビット7のSLはスラントすなわちグラフィック文字を傾斜させるかどうかのフラグです。ビット6のRは四辺形描画かどうかのフラグです。ビット5のCはサークル，すなわち円（弧）の描画かどうかのフラグです。ビット4のTはテキスト，すなわちグラフィックス文字描画かどうかのフラグです。ビット3のLはライン，すなわち直線かどうかのフラグです。

これらのフラグを設定することにより，GDCは直線描画なのか，BOXかグラフィックス文字かなどを判断します。フラグの組み合わせと描画種類の対応は図4-10のようになります。ビット2～0のDIRは描画方向です。GDCは多くのアルゴリズムがそうであるように，基本的には0°～45°の間しか描画できません（円なら1/8円弧，直線なら0°～45°の角度の直線しか描けません）。それを描画方向を変えることによって，360°どの方向にでも引けるようにしているわけです。このDIRと実際の描画方向は図4-10のようになります。

ふたたび図4-9にもどります。

VECTW 1コマンドの第2パラメータはGDCの内部にある描画パラメータレジスタDCの下位8ビットを送出しています。

これまでも述べたように，GDCには内部に描画パラメータレジスタと呼ばれる，14ビット長のレジスタがあります。このレジスタの設定値により，直線

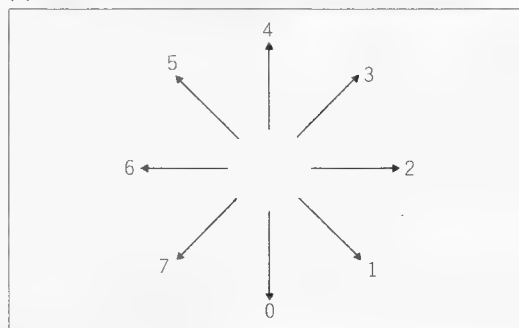
図4-10

SL	R	C	T	L	
0	0	0	0	0	◦キャラクタモード等描画 ◦ドット描画 ◦リードコマンド実行時 ◦DMA
0	0	0	0	1	ライン
0	0	0	1	0	グラフィックス文字描画
0	0	1	0	0	円(弧)
0	1	0	0	0	ボックス
1	0	0	1	0	スラント文字

の長さや方向がGDCに知らされます。最大4096×4096のGDCのVRAM空間ですべての方向,長さを指定できるように14ビット構成となっています。GDCとCPUのやりとりは8ビットを基本としているのでDCレジスタ値を送るのに,このように下位8ビット,上位6ビットに分けて行います。

第3パラメータのビット6にDGDというビットがあります。これはDynamic Graphic Drawingビットと呼ばれます。GDCには動作モードが3つあり,それぞれキャラクタモード,キャラクタ・グラフィック混在モード,グラフィックモードと呼ばれています。ちなみに,PC-9801にはGDCが2個使われていますが,マスタ動作しているGDCはキャラクタモード,スレーブ動作しているほうはグラフィックモードで使われているようです。

そこでDCDですが,これはキャラクタ・グラフィック混在モード時にグラ

図4-11 DIRと実際の描画方向

フィック動作をさせるために立てるフラグです。通常は必ず0にしておかないとGDCは妙な動作をします。

第3パラメータの残り6ビットはDCレジスタのHIGH6ビットを設定します。

第4，第5パラメータはDDレジスタの LOW, HIGH

第6，第7パラメータはD2 レジスタの LOW, HIGH

第8，第9パラメータはD1 レジスタの LOW, HIGH

をそれぞれ送出しています。ワード長のパラメータが多いのでワード長のパラメータ送出ルーチンを作ったと述べましたが，それはこれらDD, D2, D1, DMレジスタの送出に使っていたわけです。

BOX描画させるには

さてBOXですが，BOX描画をさせるためには図4-10からもわかるように，SL, R, C, T, Lの各フラグを0 1 0 0 0にセットすればよいのです。DIRはデータをうまく操作すれば，DIR=0であらゆるBOXは描けます。結局，VECTWコマンドの第1パラメータは

0 1 0 0 0 0 0　すなわち40Hとなります。

データを操作するというのは実に簡単なことで

$$\begin{cases} x_1=\min(x_1,\ x_2) \\ y_1=\min(y_1,\ y_2) \end{cases} \qquad \begin{cases} x_2=\max(x_1,\ x_2) \\ y_2=\max(y_1,\ y_2) \end{cases}$$

とBOXの左上の座標を（x_1, y_1）に，右上の座標を（x_2, y_2）にすればよいのです。これでDIRを常に0としてBOXが描けます。

この操作を行ったあと，各描画パラメータレジスタ値をセットします。BOXのときは

$DC=3$　　$D=y_2-y_1$

$D2=x_2-x_1$　　$D_1=-1$

$DM=y_2-y_1$

をセットすればよいのです。

一般にGDCでグラフィック描画させるためには，これまで何度も述べたように次のような方法をとります。

1. TEXTW コマンド　パラメータ送出

2. WRITE コマンド　送出
3. CSRW コマンド　パラメータ送出
4. VECTW コマンド　パラメータ送出
5. VECTE コマンド　送出

線パターン（破線，実線，一点鎖線）や描画モード（AND, OR, XOR など）を変更する必要がなければ，1，2を省略してもよいのです。

コマンドを解説する

少しつっこんで解説してみましょう。

1のTEXTWコマンドは実線，破線，一点鎖線などを指定するコマンドです。これはむろん，円描画時にも可能で，前出のサークルルーチンでTEXTWのパラメータFFFFHを3333Hなどに書き換えると「破線の円」！が描けます。TEXTWコマンドはGDC内部のデータ内部に線データパターン(PTN)を格納します。グラフィック描画時にはパラメータは図4-12のようになっています。TEXTWコマンドとして，これまで78Hを使ってきましたが，厳密にはTEXTWコマンドは次のようになっています。

TEXTW　0 1 1 1 1 ←RA→

RA'とはGDC内部のRAMのアドレスです。0番目から入れるには，RA'=0とすればよいので

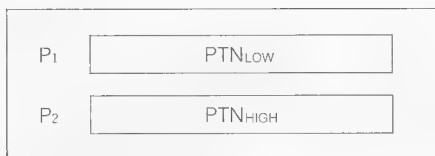
0 1 1 1 1 0 0 0 =78H

となります。線パターンのHIGHバイトだけ変更するには

RA'=1

として

0 1 1 1 1 0 0 1 =79H

図4-12

をコマンド送出し，HIGH バイトを送出（パラメータ送出）すればよいのです。もっともそんな必要はあまりないでしょうが。

TEXTW コマンドはグラフィック描画時より，グラフィックス文字描画時には大きな力をもってくるのですが，それはあとで解説します。BOX ルーチンに関しては TEXTW コマンドは78H，描画パターン PTN=FFFFH として実線描画しました。もちろん，FFFFH を変更すれば破線の BOX も描けます。

2 の WRITE コマンドはグラフィックス描画時にはドット修正モードの選択を行います。GDC は描画時にリード・モディファイ・ライトを行います。これはどういうことかというと，描画するときに描画される位置の VRAM データをまず GDC が読み込み（リード），描画するデータと OR，XOR などをとり（モディファイ），VRAM にふたたび書く（ライト）ことです。

このモディファイ時に何をするかを WRITE コマンドで指定します。グラフィックス描画時，WRITE コマンドは23H を使ってきましたが，WRITE コマンドは厳密には

WRITE　0　0　1 ←WLH→ 0 ←MOD→

となっています。WLH は別の機会に説明しますが WLH=00，MOD が問題のモディファイのモードを決めるビットです。MOD の対応は図4-13のようになります。

MOD=00　REPLACE

は元の VRAM のデータと描画パターンを置き換えます。

図4-13

MOD		モディファイ
0	0	REPLACE
0	1	COMPLEMENT
1	0	CLEAR
1	1	SET

MOD=01　COMPLEMENT

は描画パターンが1であるドットがVRAMでも1であれば0にします。VRAMで0ならば1にします。

MOD=10　CLEAR

は描画パターンが1であるドットがVRAMでも1ならば0にします（線を消すときに使います)。

MOD=11　SET

は描画パターンが1であるドットがVRAMで0ならば1にします（線を描くときに使います)。

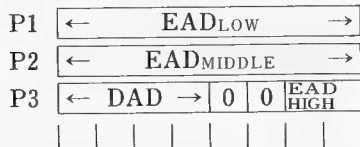
少し分かりにくいでしょうが，グラフィックス文字の説明のときにもっと具体的に考えることにします。

BOX描画時には，MOD=00またはMOD=11とします。消去にはMOD=10とします。それは

　0 0 1 0 0 0 0 0 0　=20H

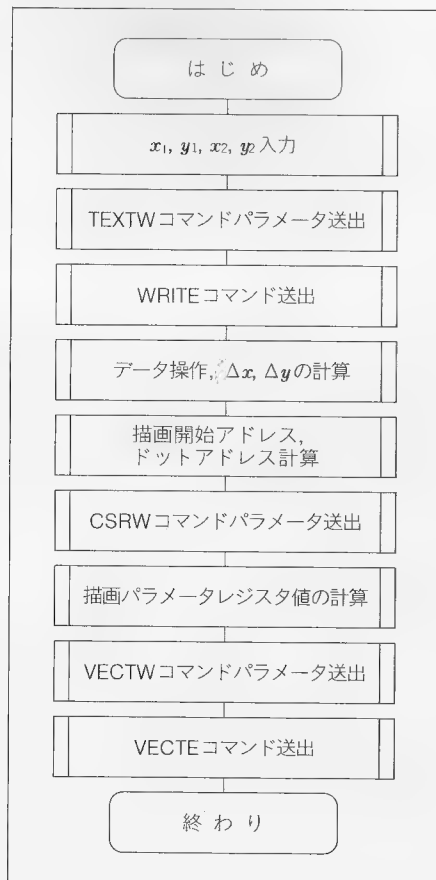
を使います。

3のCSRWコマンドは描画開始位置をGDCに知らせます。描画開始位置とは描画開始アドレスEADとドットアドレスDADです。CSRWのパラメータも，キャラクタモードかグラフィックモードかなどで変わってきます。グラフィックモード時にはパラメータは次のようになります。

P1	←			EAD LOW				→
P2	←			EAD MIDDLE				→
P3	←	DAD			→	0	0	EAD HIGH

GDCは最大2048×2048のVRAMを扱えるため，2048×2048/16=2^{18}のアドレス空間があります。そのためにEADも18ビット長となります。またGDCは1アドレス16ビットですからDADは4ビット（2^4=16）となります。

PC-9801の場合，GDCから見たVRAMは青4000H～，赤8000H～，緑C000H～なので，EADのHIGHは必ず0です。何度もいっていることですが，CPUから見たVRAMとGDCからVRAMはアドレスがまったく違います（物理的には同じRAM)。気をつけてください。

4のVECTWは説明しました。5のVECTEはGDCに描画開始を指令す

図4-14

るもので，コマンドは

6CH

以上の1～5のステップを踏まえてBASICプログラムするとリスト4-4のようになります。フローチャートは図4-14。

これでBOXの描き方は分かったでしょう。例によってこれをアセンブラに落としておきます。いつもいうようにアセンブラなりコンパイラを使わないとGDCを直接ドライブした意味がありません。BOXによるデモプログラムもい

っしょに入れておきました（リスト4-5)。これはBASICではリスト4-6のようになるでしょう。速度を比較してほしいのですが，BASICでは10秒，アセンブラではあっという間です。

GDCの各コマンドを少し深く掘り下げてみました。これでLINE, CIRCLE, BOXを終わったわけで，これだけでもかなりのことができると思います。

ボックスルーチンはいかがでしたか。いつもながら，GDCをアセンブラでドライブしたときの速度には驚きます。

リスト4-4 BASICによるBOXルーチンの例

```
1000 '
1010 '
1020 '
1030 '
1040 '
1050 *BOX.ROUTINE
1060   GOSUB *INPUT.PARAMETER
1070   GOSUB *TEXTW.ROUTINE
1080   GOSUB *WRITE.ROUTINE
1090   GOSUB *SET.DX.AND.DY
1100   GOSUB *CUL.EAD.AND.DAD
1110   GOSUB *CSRW.ROUTINE
1120   GOSUB *SET.DRAWING.PARAMETER.REGISTER
1130   GOSUB *VECTW.ROUTINE
1140   GOSUB *VECTE.ROUTINE
1150 END
1160 '
1170 *INPUT.PARAMETER INPUT "BOX(X1,Y1)-(X2,Y2) ";X1,Y1,X2,Y2:RETURN
1180 '
1190 *TEXTW.ROUTINE
1200   COMMAND=&H78 : GOSUB *OUT.COMMAND
1210   PARAMETER.W=&HFFFF : GOSUB *OUT.WORD.PARAMETER
1220 RETURN
1230 '
1240 *WRITE.ROUTINE COMMAND=&H20 : GOSUB *OUT.COMMAND:RETURN
1250 '
1260 *SET.DX.AND.DY
1270   IF X1>X2 THEN SWAP X1,X2
1280   IF Y1>Y2 THEN SWAP Y1,Y2
1290   DX=X2-X1 : DY=Y2-Y1
1300 RETURN
1310 '
1320 *SET.DRAWING.PARAMETER.REGISTER
1330   DC.REG=3 : D.REG=DY
1340   D2.REG=DX : D1.REG=-1 : DM.REG=DY
1350 RETURN
1360 '
1370 '
1380 *CUL.EAD.AND.DAD
1390    EAD=&H4000+40*Y1+X1 ¥ 16 : DAD=X1 MOD 16
1400 RETURN
1410 '
1420 *CSRW.ROUTINE
1430   COMMAND=&H49:GOSUB *OUT.COMMAND
```

```
1440   PARAMETER.W=EAD : GOSUB *OUT.WORD.PARAMETER
1450   PARAMETER=DAD*16 : GOSUB *OUT.PARAMETER
1460 RETURN
1470 '
1480 *VECTW.ROUTINE
1490   COMMAND=&H4C:GOSUB *OUT.COMMAND
1500   PARAMETER=&H40 : GOSUB *OUT.PARAMETER
1510   PARAMETER.W=DC.REG : GOSUB *OUT.WORD.PARAMETER
1520   PARAMETER.W=D.REG  : GOSUB *OUT.WORD.PARAMETER
1530   PARAMETER.W=D2.REG : GOSUB *OUT.WORD.PARAMETER
1540   PARAMETER.W=D1.REG : GOSUB *OUT.WORD.PARAMETER
1550   PARAMETER.W=DM.REG : GOSUB *OUT.WORD.PARAMETER
1560 RETURN
1570 '
1580 *VECTE.ROUTINE COMMAND=&H6C : GOSUB *OUT.COMMAND : RETURN
1590 '
1600 *OUT.COMMAND WHILE INP(&HA0) AND 2:WEND:OUT &HA2,COMMAND:RETURN
1610 '
1620 *OUT.PARAMETER WHILE INP(&HA0) AND 2:WEND:OUT &HA0.PARAMETER:RETURN
1630 '
1640 *OUT.WORD.PARAMETER
1650  PARAMETER=VAL("&H"+RIGHT$("000"+HEX$(PARAMETER.W),2))
1660  GOSUB *OUT.PARAMETER
1670  PARAMETER=VAL("&H"+LEFT$(RIGHT$("000"+HEX$(PARAMETER.W),4),2))
1680  GOSUB *OUT.PARAMETER
1690 RETURN
```

リスト4-5

```
;
; BOX COMMAND DEMO
;

;
; CONSTANTS FOR BOX COMMAND DEMO


STATUS_READ_PORT          EQU     0A0H  グラフィック画面用GDCのステータスリードボード=A0H
COMMAND_OUT_PORT          EQU     0A2H  グラフィック画面用GDCのコマンド出力用ポート=A2H
PARAMETER_OUT_PORT        EQU     0A0H  グラフィック画面用GDCのパラメーター出力用ポート=A0H


        CSEG    0       これ以降がコードセグメント0000Hに属すことを示す
        ORG     0A000H  アセンブル開始アドレス = A000H

BOX_DEMONSTRATION:      BOXルーチンを使ったデモ
;                       (正方形を4つずつ場所と大きさを変えながら描画)
;                                             { データセグメントを0に,
        XOR     AX,AX           ! MOV   DS,AX { (CP/M86のユーザーは不用)
        CALL    START_DISPLAY                 {  グラフィック画面の表示開始
        CALL    SET_640_400_COLOR_MODE  グラフィック画面のモードを640×400カラーモードに
        MOV     INDEX,2         ! MOV   COLOR,0  変数INDEX = 2, COLOR = 0
BOX_DEMONSTRATION_LOOP:
        MOV     AX,320          ! ADD   AX.INDEX } X1 = 320 + INDEX
        MOV     X1,AX
        ADD     AX,INDEX        ! MOV   X2,AX      X2 = X1 + INDEX
        MOV     DX,INDEX                         } DXレジスタにINDEX/4をセット
        SAR     DX,1            ! SAR   DX,1
        MOV     AX,200          ! ADD   AX,DX    } Y1 = 200 + INDEX/4
        MOV     Y1,AX
```

```
        ADD     AX,INDEX          ! MOV    Y2,AX          Y2 = Y1 + INDEX
        PUSH    DX       ! CALL   BOX     ! POP    DX     DXレジスタを保存しながら
                                                          BOXルーチンをコール
        MOV     AX,320            ! SUB    AX,INDEX     } X1 = 320 - INDEX
        MOV     X1,AX                                   }
        SUB     AX,INDEX          ! MOV    X2,AX          X2 = X1 - INDEX
        MOV     AX,200            ! ADD    AX,DX        } Y1 = 200 + INDEX 4
        MOV     Y1,AX                                   }   (∵DX = INDEX 4 )
        ADD     AX,INDEX          ! MOV    Y2,AX          Y2 = Y1 + INDEX
        PUSH    DX       ! CALL   BOX     ! POP    DX     DXレジスタを保存しつつ,
                                                          ボックスルーチンをコール
        MOV     AX,320   ! ADD    AX,INDEX              } X1 = 320 + INDEX
        MOV     X1,AX                                   }
        ADD     AX,INDEX          ! MOV    X2,AX          X2 = X1 + INDEX
        MOV     AX,200   ! SUB    AX,DX                 } Y1 = 200 - INDEX 4
        MOV     Y1,AX                                   }
        SUB     AX,INDEX          ! MOV    Y2,AX          Y2 = Y1 - INDEX
        PUSH    DX       ! CALL   BOX     ! POP    DX     DXレジスタを保存しながら
                                                          ボックスルーチンをコール
        MOV     AX,320            ! SUB    AX,INDEX     } X1 = 320 - INDEX
        MOV     X1,AX                                   }
        SUB     AX,INDEX          ! MOV    X2,AX          X2 = X1 - INDEX
        MOV     AX,200   ! SUB    AX,DX                 } Y1 = 200 - INDEX 4
        MOV     Y1,AX                                   }
        SUB     AX,INDEX          ! MOV    Y2,AX          Y2 = Y1 - INDEX
        PUSH    DX       ! CALL   BOX     ! POP    DX     DXを保存してBOX
                                                          ルーチンをコール
        MOV     AL,COLOR          ! INC    AL           } COLOR = COLOR + 1
        MOV     COLOR,AL                                }
        CMP     AL,3              ! JNE    MAIN1        } もし  COLOR = 3  ならば
        MOV     COLOR,0                                 } COLOR = 0


MAIN1:
        MOV     AX,INDEX          ! INC    AX           } INDEX = INDEX + 1
        MOV     INDEX,AX                                }
        CMP     AX,158   ! JG     MAIN2                   もし  INDEX > 158  ならば
        JMP     BOX_DEMONSTRATION_LOOP                    MAIN 2へメインループへ
MAIN2:
        IRET    おわり(BASICに戻る)


START_DISPLAY:                                          | CP M-86では最初グラフィック
;                                                       | 画面を消しているので, ディス
; GRAPHIC SCREEN IS OBSCURED BY CP/M-86                 | プレイ開始を指令するルーチン
;                                                       |
; CALL START DISPLAY ROUTINE IN ROM                     |
;
        MOV     AH,40H   ! INT    18H                   } 内部ルーチンを呼んでSCREEN,
        RET                                             } 0と同様のことをさせている

SET_640_400_COLOR_MODE:
;

        MOV     AH,42H   ! MOV    CH,0C0H               | 同様に内部ルーチンを呼んで
        INT     18H                                     |   SCREEN 3と同様のこと
        RET                                             |   をさせている。



; BOX MAIN ROUTINE   | 今回の主題BOXのメインルーチン
                     | 入力(X1, Y1)-(X2, Y2), COLOR(=0 : 青, =1 : 赤, =2 : 緑)
```

```
BOX:
        CALL    TEXTW_ROUTINE            TEXTWコマンド，パラメータをGDCに送出
        CALL    WRITE_ROUTINE                    WRITEコマンドをGDCに送出
        CALL    SET_DX_AND_DY          X1, Y1, X2, Y2に応じてΔX, ΔYを求める
        CALL    CUL_EAD_AND_DAD   描画開始アドレスEAD, ドットアドレスDADを計算
        CALL    CSRW_ROUTINE               CSRWコマンド・パラメータをGDCに送出
        CALL    SET_DRAWING_PARAMETER_REGISTER  描画パラメータレジスタ値の計算
        CALL    VECTW_ROUTINE             VECTWコマンド・パラメータをGDCに送出
        CALL    VECTE_ROUTINE          VECTEコマンドをGDCに送出→BOX描画開始
        RET     おわり

TEXTW_ROUTINE:    TEXTWコマンドパラメータをGDCに送出するルーチン

        MOV     COMMAND,78H        ! CALL   OUT_COMMAND
        MOV     WORD_PARAMETER,0FFFFH   ! CALL   OUT_WORD_PARAMETER
        RET     おわり        ワード長のパラメータFFFFHをパラメータ出力
                                          (実線)
WRITE_ROUTINE:
                              WRITEコマンドをGDCに送出するルーチン
        MOV     COMMAND,20H        ! CALL   OUT_COMMAND
        RET     おわり                   20H(＝WRITE)をGDCへコマンド出力

SET_DX_AND_DY:              ΔX, ΔYをX1, Y1, X2, Y2に対応して計算するルーチン

        MOV     AX,X1
        CMP     AX,X2     ! JLE    SET1    } もしX1＞＝X2ならば
        XCHG    X2,AX     ! MOV    X1,AX   } X1とX2をいれかえる
SET1:
        MOV     AX,Y1
        CMP     AX,Y2     ! JLE    SET2    } もしY1＞＝Y2ならば
        XCHG    Y2,AX     ! MOV    Y1,AX   } Y1とY2をいれかえる
SET2:
        MOV     AX,X2     ! SUB    AX,X1   } ΔX＝X2－X1
        MOV     DELTA_X,AX                 }
        MOV     AX,Y2     ! SUB    AX,Y1   } ΔY＝Y2＝Y1
        MOV     DELTA_Y,AX                 }
        RET     おわり

SET_DRAWING_PARAMETER_REGISTER:    描画パラメータレジスタ値の計算

        MOV     DC_REG,3                                DC＝3
        MOV     AX,DELTA_Y         ! MOV    D_REG,AX    D＝ΔY
        MOV     AX,DELTA_X         ! MOV    D2_REG,AX   D2＝ΔX
        MOV     D1_REG,-1                               D1＝－1
        MOV     AX,DELTA_Y         ! MOV    DM_REG,AX   DM＝ΔY
        RET     おわり




CUL_EAD_AND_DAD:    描画開始アドレス，ドットアドレスの計算

        CMP     COLOR,0 ! JNE     CUL1           } COLOR＝0なら
        MOV     BX,4000H          ! JMPS  CUL2   }    VRAM START＝4000H
CUL1:
        CMP     COLOR,1           ! JNE   CUL3   } COLOR＝1なら
        MOV     BX,8000H          ! JMPS  CUL2   }    VRAM START＝8000H
CUL3:
                                    COLOR＝2なら VRAM START＝C000H
        MOV     BX,0C000H
CUL2:
        MOV     AX,X1                            }
        MOV     CL,4      ! SHR    AX,CL         } AX＝X1 ¥16(シフトで割り算を行っている)
```

```
        ADD     BX,AX                               BX = VRAM START ADRESS + X1 ¥16
        MOV     AX,40    ! MOV    CX,Y1       }  AX = 40 * Y1
        MUL     CX
        ADD     BX,AX    ! MOV    EAD,BX         EAD = VRAM START + 40 * Y1 + X1 ¥16

        MOV     AX,X1    ! AND    AL,15       }  DAD = X1MOD16
        MOV     DAD,AL
        RET     終わり

CSRW_ROUTINE:       CSRWコマンド・パラメータをGDCに送出するルーチン

        MOV     COMMAND,49H      ! CALL   OUT_COMMAND        49H(= CSRW)をGDCにコマンド送出
        MOV     AX,EAD                                        } EADをGDCに
        MOV     WORD_PARAMETER,AX         ! CALL  OUT_WORD_PARAMETER } パラメータ出力
        MOV     AL,DAD   ! MOV    CL,4                          (ワード長)
        SHL     AL,CL                                   } AL = DAD * 16
        MOV     PARAMETER,AL     ! CALL   OUT_PARAMETER  DAD*16をGDCにパラメータ出力
        RET     終わり

VECTW_ROUTINE:      VECTWコマンド，パラメータGDCに送出するルーチン
                                                  4CH(= VECTW) をGDCにコマンド出力
                                                  40H(BOX指定 DIR = 0)を
        MOV     COMMAND,4CH      ! CALL   OUT_COMMAND     GDCにパラメータ出力
        MOV     PARAMETER,40H    ! CALL   OUT_PARAMETER
        MOV     AX,DC_REG        ! MOV    WORD_PARAMETER,AX  } DCをGDCにワード長
        CALL    OUT_WORD_PARAMETER                              パラメータ出力
        MOV     AX,D_REG         ! MOV    WORD_PARAMETER,AX  } DをGDCにワード長
        CALL    OUT_WORD_PARAMETER                              パラメータ出力
        MOV     AX,D2_REG        ! MOV    WORD_PARAMETER,AX  } D2をGDCにワード長
        CALL    OUT_WORD_PARAMETER                              パラメータ出力
        MOV     AX,D1_REG        ! MOV    WORD_PARAMETER,AX  } D1をGDCにワード長
        CALL    OUT_WORD_PARAMETER                              パラメータ出力
        MOV     AX,DM_REG        ! MOV    WORD_PARAMETER,AX  } DMをGDCにワード長
        CALL    OUT_WORD_PARAMETER                              パラメータ出力
        RET     終わり

VECTE_ROUTINE:       VECTEコマンドをGDCに送出するルーチン
                     (描画開始)
        MOV     COMMAND,6CH      ! CALL   OUT_COMMAND  6CH(= VECTE)をGDO
        RET     終わり                                 にコマンド出力

OUT_COMMAND:         GDCにコマンドを送出するルーチン

OUT_CL: IN      AL,STATUS_READ_PORT           } GDCのステータスを読みFIFOがFULLでなく
        AND     AL,2     ! JNZ    OUT_CL      }   なるのをまつ
        MOV     AL,COMMAND       ! OUT    COMMAND_OUT_PORT,AL :コマンド出力ポート
        RET     終わり                                          よりCOMMANDを送出

OUT_PARAMETER:       GDCにパラメータを送出するルーチン
;

OUT_PL: IN      AL,STATUS_READ_PORT           } FIFOがFULLでないのを確認
        AND     AL,2     ! JNZ    OUT_PL      }
        MOV     AL,PARAMETER     ! OUT    PARAMETER_OUT_PORT,AL  パラメータ出力
        RET                                                     よりPARAMETERを送出

OUT_WORD_PARAMETER:        ワード長のパラメータをGDCに送出するルーチン

        MOV     BX,WORD_PARAMETER
        MOV     PARAMETER,BL     ! CALL   OUT_PARAMETER  WORD PARAMETERの下位バイト送出
        MOV     PARAMETER,BH     ! CALL   OUT_PARAMETER  WORD PARAMETERの上位バイト送出
        RET     終わり
```

```
        DSEG    0
        ORG     0B000H
X1      DW      0                ! Y1     DW      0 } BOXルーチンの引数
X2      DW      0                ! Y2     DW      0 }
DELTA_X DW      0                ! DELTA_Y        DW      0
COLOR   DB      0

DC_REG  DW      0       !        D_REG    DW      0 }
D2_REG  DW      0       !        D1_REG   DW      0 } 描画パラメータレジスタ
DM_REG  DW      0                                   }
EAD     DW      0       ! DAD    DB       0
COMMAND DB      0                ! PARAMETER      DB      0
WORD_PARAMETER  DW      0
INDEX   DW      0


        END
```

リスト4-6

```
1000 SCREEN 3,1
1010 TIME$="00:00:00"
1020 FOR I=2 TO 158:K=I¥2:COLOR.NUMBER=2^(I MOD 3)
1030 X=320+I:LINE(X,200+I/4)-STEP(I,I).COLOR.NUMBER,B
1040 X=320-I:LINE(X,200+I/4)-STEP(-I,I),COLOR.NUMBER,B
1050 X=320+I:LINE(X,200-I/4)-STEP(I,-I),COLOR.NUMBER,B
1060 X=320-I:LINE(X,200-I/4)-STEP(-I,-I),COLOR.NUMBER,B
1070 NEXT
1080 PRINT TIME$
```

アセンブラのない人のためのダンプリスト

```
A000--33 C0 8E D8 E8 CB 00 E8 CD 00 C7 06 1E B0 02 00
A010--C6 06 0C B0 00 B8 40 01 03 06 1E B0 A3 00 B0 03
A020--06 1E B0 A3 04 B0 8B 16 1E B0 D1 FA D1 FA B8 C8
A030--00 03 C2 A3 02 B0 03 06 1E B0 A3 06 B0 52 E8 9D
A040--00 5A B8 40 01 2B 06 1E B0 A3 00 B0 2B 06 1E B0
A050--A3 04 B0 B8 C8 00 03 C2 A3 02 B0 03 06 1E B0 A3
A060--06 B0 52 E8 78 00 5A B8 40 01 03 06 1E B0 A3 00
```

動かし方
MON⏎
] G0⏎
の後，左のダンプを打ち込み
CTRL-B で
BASIC にぬけて

```
A070--B0 03 06 1E B0 A3 04 B0 B8 C8 00 2B C2 A3 02 B0
A080--2B 06 1E B0 A3 06 B0 52 E8 53 00 5A B8 40 01 2B
A090--06 1E B0 A3 00 B0 2B 06 1E B0 A3 04 B0 B8 C8 00
A0A0--2B C2 A3 02 B0 2B 06 1E B0 A3 06 B0 52 E8 2E 00
A0B0--5A A0 0C B0 FE C0 A2 0C B0 3C 03 75 05 C6 06 0C
A0C0--B0 00 A1 1E B0 40 A3 1E B0 3D 9E 00 7F 03 E9 44
A0D0--FF CF B4 40 CD 18 C3 B4 42 B5 C0 CD 18 C3 E8 16
A0E0--00 E8 25 00 E8 2B 00 E8 7C 00 E8 B5 00 E8 57 00
A0F0--E8 CE 00 E8 09 01 C3 C6 06 1A B0 78 E8 09 01 C7
A100--06 1C B0 FF FF E8 18 01 C3 C6 06 1A B0 20 E8 F7
A110--00 C3 A1 00 B0 3B 06 04 B0 7E 07 87 06 04 B0 A3
A120--00 B0 A1 02 B0 3B 06 06 B0 7E 07 87 06 06 B0 A3
A130--02 B0 A1 04 B0 2B 06 00 B0 A3 08 B0 A1 06 B0 2B
A140--06 02 B0 A3 0A B0 C3 C7 06 0D B0 03 00 A1 0A B0
A150--A3 0F B0 A1 08 B0 A3 11 B0 C7 06 13 B0 FF FF A1
A160--0A B0 A3 15 B0 C3 80 3E 0C B0 00 75 05 BB 00 40
A170--EB 0F 80 3E 0C B0 01 75 05 BB 00 80 EB 03 BB 00
A180--C0 A1 00 B0 B1 04 D3 E8 03 D8 B8 28 00 8B 0E 02
A190--B0 F7 E1 03 D8 89 1E 17 B0 A1 00 B0 24 0F A2 19
A1A0--B0 C3 C6 06 1A B0 49 E8 5E 00 A1 17 B0 A3 1C B0
A1B0--E8 6D 00 A0 19 B0 B1 04 D2 E0 A2 1B B0 E8 54 00
A1C0--C3 C6 06 1A B0 4C E8 3F 00 C6 06 1B B0 40 E8 43
A1D0--00 A1 0D B0 A3 1C B0 E8 46 00 A1 0F B0 A3 1C B0
A1E0--E8 3D 00 A1 11 B0 A3 1C B0 E8 34 00 A1 13 B0 A3
A1F0--1C B0 E8 2B 00 A1 15 B0 A3 1C B0 E8 22 00 C3 C6
A200--06 1A B0 6C E8 01 00 C3 E4 A0 24 02 75 FA A0 1A
A210--B0 E6 A2 C3 E4 A0 24 02 75 FA A0 1B B0 E6 A0 C3
A220--8B 1E 1C B0 88 1E 1B B0 E8 E9 FF 88 3E 1B B0 E8
A230--E2 FF C3 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
DEF SEG = 0 ⏎
A = & HA000 ⏎
CALL A ⏎
```

最近16ビットのパソコンが，かなり見受けられますが，画面をCPUで描画しているものが大半であり，8ビットパソコンと大差ない速度しか出ていない機種も多いのです。96Kとか，128KのVRAMをもつことがなかば常識となっている現在，この広いVRAMエリアの描画をすべてCPUにさせるのはいくら16ビットといえども荷が重いでしょう。またグラフィック描画にCPUの処理が費やされて，全体的にスループットが下がります。「これで本当に16ビットか」と思うことも多いのです。やはりこれからは描画プロセッサはぜひとも搭載させてほしいものです。日電のほかに，トムソン，日立なども描画プロセッサを発表しているので，先が楽しみです。

グラフィックス文字描画のプログラム

ここでグラフィック文字描画を行います。PC-9801ではキャラクタは通常キャラクタ画面に出力されます。このようにグラフィック画面とキャラクタ画面が分離される構成は，非常に使いやすいのですが，どうしてもグラフィック画面に文字を出したいときもあるでしょう。それを可能にするのが，このグラフィックス文字描画機能です。GDCのグラフィックス文字描画機能は8×8ドットのグラフィックス文字描画と8×8以外のグラフィックス文字描画がありますが，ここでは8×8のグラフィックス文字描画を解説します。例によって①アルゴリズム，②BASICによる例，③アセンブラによる例，と三段構えで説明します。GDCはやはりアセンブラでドライブしないとありがたさが半減します。

ズーム機能

知っている方も多いと思いますが，GDCには2つのズーム機能があります。つまり
(1)拡大表示機能（ズームリード機能）
(2)拡大描画機能（ズームライト機能）
の2つです。(1)の拡大表示機能は図形やグラフィックス文字を描画したあとで2倍，3倍と拡大して表示させる機能です。表示する際に拡大されるので，VRAM内のデータは小さいままです。それに対し，(2)の拡大描画機能は描画するときに2倍，3倍と拡大して，VRAMに書き込む機能です。

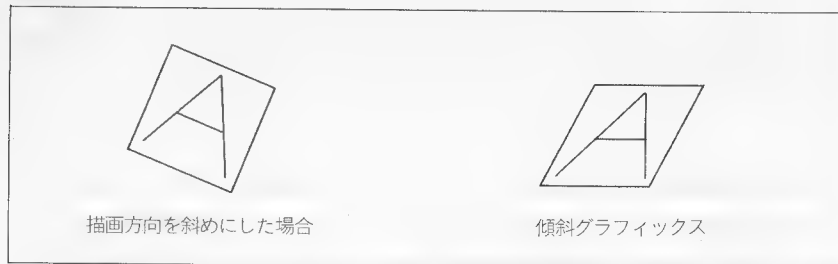
これまでも何度かいってきましたが，9801では拡大表示機能は使えないようです。このズームリード機能はGDCの機能ではありますが，シフトクロックをズーム係数に従って変化させる回路がほかに必要となります。9801ではこの機能はサポートしていないようです。このズームリード機能は，拡大して図形の一部をライトペンで修正して元に戻したり，デジタイズした画面を拡大したりでき，非常におもしろい機能なので，サポートされていないのは残念です。グラフィックターミナル（たとえばAED512）ではおなじみの機能で，知っている方も多いでしょう。

この拡大表示に対し，拡大描画は PC-9801 でも可能です。書き込み時に拡大するので9801でもできるのは当然ですが，このズームライトも，ズームリードに劣らずおもしろい機能です。

グラフィックス文字描画時にズームライトすると，ちょうど FM-8 の Symbol 文のようになります。ズームと描画方向を変えることにより，45°おきに傾けることができる Symbol 機能が簡単に作れます。

傾斜グラフィックス文字

描画方向を変えることによって，GDC では，グラフィックス文字を斜めにすることができますが，これはそうではなく字体を上にいくほど横にずらす機能です（図4-15)。

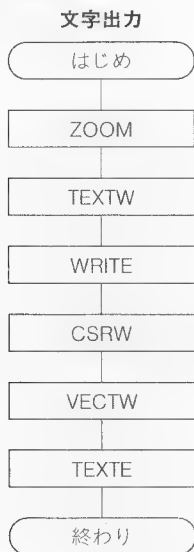
図4-15

この機能により，非常に美しい文字がディスプレイされます。傾斜グラフィックスをさせるには VECTW コマンドのパラメータの S (Slant) フラグを立てるかどうかだけなので，容易に行うことができます。

グラフィックス文字描画のプログラミング

ここの主題は，「1文字出力ルーチンを作る」ことといえます。通常1文字出力ルーチンはアスキーコード（と必要とあればキャラクタ座標）を与えてコールしますが，ここのルーチンは，引数にグラフィック座標 (x, y) 文字の方向 Direction，字体 Slant，拡大係数 Zoom，アスキーコード ASCII-CODE，色を与えてコールするようになっています。

アルゴリズム

図4-16に1文字出力のフローチャートを示します。これまでのGDCのプログラム（直線，円など）と似ていますが，いくつかのコマンドが変わっています。以下それらを含めて説明していきます。

図4-16
文字出力

ZOOMコマンド

ZOOMコマンドは拡大表示時の拡大係数ZRと拡大描画時の拡大係数ZWを設定するコマンドです。コマンドとパラメータのフォーマットは

コマンド　|0 1 0 0 0 1 1 0| ＝46H

パラメータ |ZR|ZW|

となります。ZR，ZWは0で1倍，15で16倍となります。PC-9801に関する限り，ZR＝0でなければなりません（ズームリード機能がないため)。ZR≠0ならば，シフト・クロック周期が制御されないのでディスプレイが縦じま状になります。

TEXTWコマンド

もうおなじみのコマンドです。直線や円描画時には線種データ（実線，破線ほか）をGDCに送りましたが，グラフィックス文字描画時には文字フォントのパターンを送ります。8×8ドットの文字描画時はパラメータを

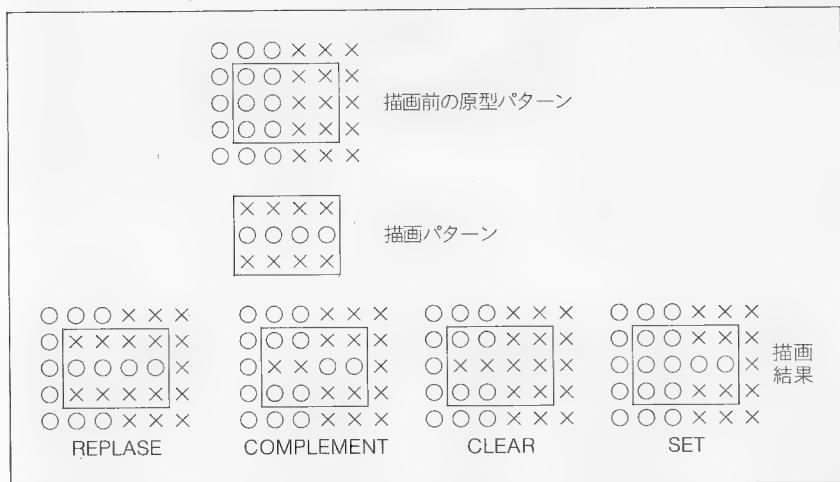
P_1	TX8
P_2	TX7
P_3	TX6
P_4	TX5
⋮	⋮
P_8	TX1

のように8回送出します。キャラクタモード時のように高速に文字を表示できない代わりに，任意のフォントが選べる利点があります。グラフィックス文字描画時にはこのように文字フォントが必要となります。当初漢字ROMからとり出そうとしていましたが，漢字ROMがないと動かないことと，任意のフォントが選べることを示すために，アセンブリ言語のプログラムでは，フォントデータは，RAM上からとってくるようにしました。一応ASCII文字はサポートしています。もちろん，このデータを自分で書き換えることもできます。

WRITEコマンド

これもおなじみのコマンドです。これから書き込む場所と，書き込むデータとをCOMPLEMENTとするかSETとするかなど，モディファイ時の動作を指定します。GDCは描画時に常にリード・モディファイ・ライト(VRAMのデータを読み，書き込むデータとの間で操作をし，VRAMに書き込む動作）を繰り返しています。ここでは，REPLACEを指定しています。指定できるモードは，

REPLACE
COMPLEMENT
CLEAR
SET

図4-17

の4つです。よくGDCのモディファイの説明に使われる図を図4-17に示します。

コマンドはそれぞれ，20H 21H 22H 23H となります。

CSRWコマンド

これもおなじみのコマンドです。直線描画時と同様に，描画実行アドレスEADと描画実行ドットアドレスDADを設定します。EAD，DADは，青色プレーンであれば，

$$\begin{cases} EAD=4000H+40y+(x \ ¥ \ 16) \\ DAD=x \bmod 16 \end{cases}$$

で求められます。

VECTWコマンド

VECTWコマンドでは描画方向，傾斜の指定を行います。8×8ドット時には，描画パラメータレジスタのDCに7を設定します。

コマンド，パラメータは次のようになります。

コマンド　|0 1 0 0 1 1 0 0|=4C

パラメータ1 | SL | R | C | T | L←DIR→ |

パラメータ2 | DCLOW |

パラメータ3 | X | DGD ←DCHIGH→ |

グラフィックス文字描画時はパラメータ1はSL=0（傾斜文字なら1）R=0, C=0, T=1, L=0, パラメータ2は7, パラメータ3は0とします。

TEXTEコマンド

直線，円などで描画開始の指令はVECTEでしましたが，グラフィック文字描画時は，TEXTEコマンドで行います。

コマンド | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | =68H

以上のコマンド・パラメータを送出すればGDCはグラフィック文字を描画してくれます。

これを，そっくりそのままBASICでプログラムするとリスト4-7のようになります。「ゐ」を出力するプログラムです。runさせると，ZOOM, SLANT, x, y, DIRECTIONをきいてくるので，まず，0, 0, 320, 200, 2を設定してみます。すると画面中心に，ゐが小さく表示されます。ZOOM=15とすると，ゐが大きく表示されます。

いつもいうとおり，GDCをBASICでドライブしても何の意味もありませ

リスト4-7

```
1000 '************************************
1010 '*                                  *
1020 '* GRAPHIC CHARACTOR DEMONSTRATION *
1030 '*                                  *
1040 '*                                  *
1050 '*                                  *
1060 '************************************
1070 '
1080 GOSUB *INPUT.DATA
1090 GOSUB *ZOOM.ROUTINE
1100 GOSUB *TEXTW.ROUTINE
1110 GOSUB *WRITE.ROUTINE
1120 GOSUB *CSRW.ROUTINE
1130 GOSUB *VECTW.ROUTINE
1140 GOSUB *TEXTE.ROUTINE
1150 END
1160 '
1170 *INPUT.DATA
1180 INPUT "ZOOM(0-15) ";ZOOM
1190 INPUT "SLANT(0:NO 1:YES) ";SLANT
1200 INPUT "X,Y ";X,Y
1210 INPUT "DIRECTION ";DIRECTION
1220 RETURN
```

```
1230 '
1240 *ZOOM.ROUTINE
1250  COMMAND=&H46:GOSUB *OUT.COMMAND
1260 PARAMETER=ZOOM:GOSUB *OUT.PARAMETER
1270 RETURN
1280 '
1290 *TEXTW.ROUTINE
1300 COMMAND=&H78 : GOSUB *OUT.COMMAND
1310  FOR C=1 TO 8:READ PARAMETER:GOSUB *OUT.PARAMETER:NEXT
1320 RETURN
1330 '
1340 *WRITE.ROUTINE
1350 COMMAND=&H20:GOSUB *OUT.COMMAND
1360 RETURN
1370 '
1380 *CSRW.ROUTINE
1390 COMMAND=&H49:GOSUB *OUT.COMMAND
1400 EAD=&H4000+40*Y+X ¥ 16 : DAD=X MOD 16
1410 WORD.PARAMETER=EAD:GOSUB *OUT.WORD.PARAMETER
1420 PARAMETER=DAD*16 : GOSUB *OUT.PARAMETER
1430 RETURN
1440 '
1450 *VECTW.ROUTINE
1460 COMMAND=&H4C:GOSUB *OUT.COMMAND
1470 PARAMETER=&H80*SLANT+&H10+DIRECTION:GOSUB *OUT.PARAMETER
1480 WORD.PARAMETER=7:GOSUB *OUT.WORD.PARAMETER
1490 RETURN
1500 '
1510 *TEXTE.ROUTINE
1520 COMMAND=&H68:GOSUB *OUT.COMMAND
1530 RETURN
1540 *FONT
1550 DATA &H40,&HA0,&HA0,&H40,&HA8,&H90,&H68,&H00
1560 '
1570 *OUT.COMMAND WHILE INP(&HA0) AND 2:WEND:OUT &HA2,COMMAND:RETURN
1580 '
1590 *OUT.PARAMETER WHILE INP(&HA0) AND 2:WEND:OUT &HA0,PARAMETER:RETURN
1600 '
1610 *OUT.WORD.PARAMETER
1620 PARAMETER=VAL("&H"+RIGHT$("000"+HEX$(WORD.PARAMETER),2))
1630 GOSUB *OUT.PARAMETER
1640 PARAMETER=VAL("&H"+LEFT$(RIGHT$("000"+HEX$(WORD.PARAMETER),4),2))
1650 GOSUB *OUT.PARAMETER
1660 RETURN
```

```
100 ' GRAPHIC CHARACTOR DRAWING
110 '
112 SCREEN 3
115 COLOR=(1,7)
130 DEF SEG=0:A=&HA000:CALL A
140 POKE &HA016,&H4:CALL A
145 ROLL 399:POKE &HA016,2:CALL A
150 POKE &HA007,1:CALL A
160 POKE &HA007,0 : GOTO 130
```

左のプログラムは，いちばん後ろにあるダンプリストを打ち込んでから走らせる「130文字×50行」のデモです。

ん。ということでアセンブリ言語で書いたリストをリスト4-8に示します。1文字出力ルーチンと，文字列出力ルーチンおよびそのデモのプログラムです。

PRINT_ONE_CHARACTOR が，1文字出力ルーチンで，引数はグラフィック座標（x, y）COLOR（0→青，1→赤，2→緑）SLANT（0：NO，1：YES），ZOOM（0〜15），ASCII-CODE，DIRECTION（0-3）です。

PRINT-STRING は文字列出力ルーチンで文字列の先頭のセグメントベースをES，オフセットをSIに入れてコールします。文字列の最後は0になっていることが必要です。

DEMO は，このルーチンを使ったデモです。

リスト4-8

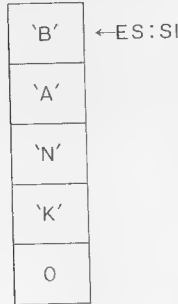
```
;
; GRAPHIC CHARACTOR DEMONSTRATION
;
;
;

STATUS_READ_PORT          EQU     0A0H     ；GDCのステータス・リードポートはA2H
COMMAND_OUT_PORT          EQU     0A2H     ；GDCのコマンド出力ポートはA2H
PARAMETER_OUT_PORT        EQU     0A0H     ；GDCのパラメータ出力ポートはA0H
CHARACTOR_FONT_ADRESS     EQU     0A200H   ；文字フォントをA200Hから置く
        CSEG    0                          ；以下コードセグメント
        ORG     0A000H                     ；A000からアセンブル
DEMO:
        CALL    INITIALIZE_ROUTINE         ；イニシャライズルーチンをコール
        MOV     SLANT,0                    ；傾斜など      (画面モード，データ)
        MOV     ZOOM,0                     ；ズーム(×1)   (セグメントの初期化)
        MOV     COLOR,0                    ；青指定
        MOV     DIRECTION,2                ；描画方向 2
        MOV     AX,0    ! MOV   ES,AX      } 文字列の先頭は0B000H
        MOV     SI,0B000H                  }
        MOV     X,20    ! MOV   Y,20       ；グラフィック座標(20,20)
        CALL    PRINT_STRING               ；文字列出力ルーチン
        IRET                               ；BASICへ戻る

PRINT_STRING:
;
;       STRING (SEGMENT ES,OFFSET SI)      ；文字列出力ルーチン
;               X,Y                        ；文字列 ES:SI から入っている(文末は0)

PRINT1:
        MOV     ES:AL,[SI]      ! INC   SI  } 文字のアスキーコードが0なら帰る
        CMP     AL,00H  ! JNE   PRINT2      }
        RET
PRINT2:
        MOV     ASCII_CODE,AL              } アスキーコードをセットして,
        PUSH    SI                         } 1文字出力ルーチンをコール
        CALL    PRINT_ONE_CHARACTOR        }
        POP     SI
        ADD     X,8
        CMP     X,631   ! JLE   PRINT3     } IF X > 631 THEN X = 0 : Y = Y + 11
        MOV     X,0     ! ADD   Y,11       }
        CMP     Y,391   ! JLE   PRINT3     } IF Y > 391 THEN Y = 8
        MOV     Y,8                        }
PRINT3:
        JMPS    PRINT1


PRINT_ONE_CHARACTOR:                       ；1文字出力ルーチン
```

```
;
; ARGUMENTS X,Y,COLOR,SLANT,ZOOM,ASCII_CODE,DIRECTION
;
        CALL    ZOOM_ROUTINE        ; ZOOMコマンドパラメータ送出
        CALL    TEXTW_ROUTINE       ; TEXTWコマンドパラメータ送出
        CALL    WRITE_ROUTINE       ; WRITEコマルンド送出
        CALL    CSRW_ROUTINE        ; CSRWコマンドパラメータ送出
        CALL    VECTW_ROUTINE       ; VECTWコマンドパラメータ送出
        CALL    TEXTE_ROUTINE       ; TEXTEコマンド送出(描画実行)
        RET
                                    ; イニシャライズルーチン
INITIALIZE_ROUTINE:                 (画面,セグメントなどをイニシャライズする)
        XOR     AX,AX               } データセグメントを0に
        MOV     DS,AX               }
        MOV     AH,40H  ! INT   18H ; ディスプレイ開始(内部ルーチン)
        MOV     AH,42H  ! MOV   CH,0C0H }
        INT     18H                     } 640×400に(内部ルーチン)
        RET                         ; 終わり

ZOOM_ROUTINE:
        MOV     COMMAND,46H     ! CALL  OUT_COMMAND     ; 46Hをコマンド出力
        MOV     AL,ZOOM                                 }
        MOV     PARAMETER,AL    ! CALL  OUT_PARAMETER   } ZOOMをパラメータ出力
        RET                                             ; 終わり

TEXTW_ROUTINE:
        MOV     COMMAND,78H     ! CALL  OUT_COMMAND ; 78Hをコマンド出力
        MOV     SI,CHARACTOR_FONT_ADRESS            ; SIにフォントの先頭をロード
        MOV     AL,ASCII_CODE   ! CBW               ; AXにアスキーコードを入れる
        MOV     CL,3    ! SHL   AX,CL;              ; AX = AX * 8
        ADD     SI,AX                               ; SI = FONT + 8 * ASCII
        MOV     BX,0                                (フォントは1文字8バイトより)
TEXTW1:
        MOV     AL,[SI+BX]          ; フォントデータの1つをALにロード
        MOV     PARAMETER,AL    ! CALL  OUT_PARAMETER ; フォントデータをパラメータ出力
        INC     BX                                    }
        CMP     BX,7    ! JLE   TEXTW1                } 8回ロードするまで繰り返す
        RET

WRITE_ROUTINE:
        MOV     COMMAND,20H     ! CALL  OUT_COMMAND     ; 20Hをコマンド出力
        RET                                             ; 終わり

CSRW_ROUTINE:
        MOV     COMMAND,49H     ! CALL  OUT_COMMAND     ; 49Hをコマンド出力
        CMP     COLOR,0 ! JNE   CSRW1                   } COLOR = 0
        MOV     BX,4000H        ! JMPS  CSRW2           }     →BX = 4000H
CSRW1:
        CMP     COLOR,1 ! JNE   CSRW3                   } COLOR = 1
        MOV     BX,8000H        ! JMPS  CSRW2           }     →BX = 8000H
CSRW3:
        MOV     BX,0C000H                               ; COLOR = 2
CSRW2:                                                  ;     →BX = C000H
        MOV     AX,Y    ! MOV   CX,40                   } BX = "VRAM開始アドレス"
        MUL     CX      ! ADD   BX,AX                   }      + 40 * Y

        MOV     AX,X    ! CWD                           } AX = X ¥16
        MOV     CX,16   ! DIV   CX                      } DX = X  MOD 16
        ADD     BX,AX

        MOV     WORD_PARAMETER,BX   ! CALL  OUT_WORD_PARAMETER ; EADを
        MOV     CL,4    ! SHL   DL,CL                          ワード出力
        MOV     PARAMETER,DL    ! CALL  OUT_PARAMETER ; DAD * 16をパラメータ出力
        RET                                           ; 終わり
```

```
VECTW_ROUTINE:
        MOV     COMMAND,4CH       ! CALL   OUT_COMMAND     ; 4CHをコマンド出力
        MOV     AL,SLANT          ! ROR    AL,1            } 第1パラメータ設定
        ADD     AL,10H   ! ADD    AL,DIRECTION             }
        MOV     PARAMETER,AL      ! CALL   OUT_PARAMETER   ; パラメータ出力
        MOV     WORD_PARAMETER,7          ! CALL    OUT_WORD_PARAMETER ; DC = 7
        RET                                                ; 終わり       をワード出力

TEXTE_ROUTINE:
        MOV     COMMAND,68H       ! CALL   OUT_COMMAND     } 68Hをコマンド出力
        RET

                                                           ; コマンド出力ルーチン
OUT_COMMAND:
        IN      AL,STATUS_READ_PORT                        } FIFOがいっぱいで
        TEST    AL,02    ! JNZ    OUT_COMMAND              } なくなるまで待つ
        MOV     AL,COMMAND        ! OUT    COMMAND_OUT_PORT,AL ; COMMANDをコマンド
        RET                                                    出力ポートより送出

OUT_PARAMETER:
        IN      AL,STATUS_READ_PORT                        } FIFOがいっぱいで
        TEST    AL,02    ! JNZ    OUT_PARAMETER            } なくなるまで待つ
        MOV     AL,PARAMETER      ! OUT    PARAMETER_OUT_PORT,AL  ; PARAMETERを
        RET                                                パラメータ出力ポートより送出

OUT_WORD_PARAMETER:                          ; ワード長のパラメータを出力するルーチン
        MOV     AX,WORD_PARAMETER
        MOV     PARAMETER,AL      ! CALL   OUT_PARAMETER   } WORD_PARAMETERの下位
        MOV     PARAMETER,AH      ! CALL   OUT_PARAMETER   }                 上位
        RET                                                  をそれぞれパラメータ出力

        DSEG    0                                          ; 以下データセグメント
        ORG     0A200H                                     ; 文字のフォントはA200Hより
TEMP    DB      040H,0A0H,0AH,040H,0A8H,090H,068H,000H; ダミーフォントデータ

        ORG     9000H                                      ; 変数は9000H以降
X       DW      0
Y       DW      0
COLOR   DB      0
SLANT   DB      0
DIRECTION       DB      0
ASCII_CODE      DB      0
ZOOM    DB      0

COMMAND DB      0
PARAMETER       DB      0
WORD_PARAMETER  DW      0

        END
```

アセンブラのない人のために，リスト4-9にダンプリストを載せておきます。文章がないとおもしろくないので，文章例を載せておきます。動かし方はリストを参照してください。横130文字縦50文字のデモはなかなか圧巻です。

線の描画のほかにキャラクタもアセンブリ言語で書けるようになったわけで，これだけでも，PC9801で十分ゲームが作れると思います。

"BANK" を出力するとき

リスト4-9

打ち込み方

MON ⏎
Co ⏎
のあと打ち込んでください。

```
A000 E8 77 00 C6 06 05 90 00 C6 06 08 90 00 C6 06 04
A010 90 00 C6 06 06 90 02 B8 00 00 8E C0 BE 00 B0 C7
A020 06 00 90 14 00 C7 06 02 90 14 00 E8 01 00 CF 8A
A030 04 46 3C 00 75 01 C3 A2 07 90 56 E8 29 00 5E 83
A040 06 00 90 08 81 3E 00 90 77 02 7E 19 C7 06 00 90
A050 00 00 83 06 02 90 0B 81 3E 02 90 87 01 7E 06 C7
A060 06 02 90 08 00 EB C8 E8 1F 00 E8 2E 00 E8 52 00
A070 E8 58 00 E8 A0 00 E8 C0 00 C3 33 C0 8E D8 B4 40
A080 CD 18 B4 42 B5 C0 CD 18 C3 C6 06 09 90 46 E8 B1
A090 00 A0 08 90 A2 0A 90 E8 B4 00 C3 C6 06 09 90 78
A0A0 E8 9F 00 BE 00 A2 A0 07 90 98 B1 03 D3 E0 03 F0
A0B0 B8 00 00 8A 00 A2 0A 90 E8 93 00 43 83 FB 07 7E
A0C0 F2 C3 C6 06 09 90 20 E8 78 00 C3 C6 06 09 90 49
A0D0 E8 6F 00 80 3E 04 90 00 75 05 BB 00 40 EB 0F 80
A0E0 3E 04 90 01 75 05 BB 00 80 EB 03 BB 00 C0 A1 02
A0F0 90 B9 28 00 F7 E1 03 D8 A1 00 90 99 B9 10 00 F7
A100 F1 03 D8 89 1E 0B 90 E8 50 00 B1 04 D2 E2 88 16
A110 0A 90 E8 39 00 C3 C6 06 09 90 4C E8 24 00 A0 05
A120 90 D0 C8 04 10 02 06 06 90 A2 0A 90 E8 1F 00 C7
A130 06 0B 90 07 00 E8 22 00 C3 C6 06 09 90 68 E8 01
A140 00 C3 E4 A0 A8 02 75 FA A0 09 90 E6 A2 C3 E4 A0
A150 A8 02 75 FA A0 0A 90 E6 A0 C3 A1 0B 90 A2 0A 90
A160 E8 EB FF 88 26 0A 90 E8 E4 FF C3 00 00 00 00 00
A200 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
A210 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
A220 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
A230 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
A240 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
A250 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

……↓以下
フォントデータ

```
A260 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
A270 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
A280 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
A290 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
A2A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
A2B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
A2C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
A2D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
A2E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
A2F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
A300 00 00 00 00 00 00 00 00 04 04 04 04 04 00 04 00
A310 0A 0A 0A 00 00 00 00 00 0A 0A 1F 0A 1F 0A 0A 00
A320 04 1E 05 0E 14 0F 04 00 03 13 08 04 02 19 18 00
A330 02 05 05 02 15 09 16 00 04 04 04 00 00 00 00 00
A340 04 02 01 01 01 02 04 00 04 08 10 10 10 08 04 00
A350 04 15 0E 04 0E 15 04 00 00 04 04 1F 04 04 00 00
A360 00 00 00 00 04 04 02 00 00 00 00 1F 00 00 00 00
A370 00 00 00 00 00 00 04 00 00 10 08 04 02 01 00 00
A380 0E 11 19 15 13 11 0E 00 04 06 04 04 04 04 0E 00
A390 0E 11 10 0C 02 01 1F 00 1F 10 08 0C 10 11 0E 00
A3A0 08 0C 0A 09 1F 08 08 00 1F 01 0F 10 10 11 0E 00
A3B0 1C 02 01 0F 11 11 0E 00 1F 10 08 04 02 02 02 00
A3C0 0E 11 11 0E 11 11 0E 00 0E 11 11 1E 10 08 07 00
A3D0 00 00 04 00 04 00 00 00 00 00 04 00 04 04 02 00
A3E0 08 04 02 01 02 04 08 00 00 00 1F 00 1F 00 00 00
A3F0 02 04 08 10 08 04 02 00 0E 11 08 04 04 00 04 00
A400 0E 11 15 1D 0D 01 1E 00 04 0A 11 11 1F 11 11 00
A410 0F 11 11 0F 11 11 0F 11 0E 11 01 01 01 11 0E 00
A420 0F 11 11 11 11 11 0F 00 1F 01 01 0F 01 01 1F 00
A430 1F 01 01 0F 01 01 01 00 1E 01 01 01 19 11 1E 00
A440 11 11 11 1F 11 11 11 00 0E 04 04 04 04 04 0E 00
A450 10 10 10 10 10 11 0E 00 11 09 05 03 05 09 11 00
A460 01 01 01 01 01 01 1F 00 11 1B 15 15 11 11 11 00
A470 11 11 13 15 19 11 11 00 0E 11 11 11 11 11 0E 00
A480 0F 11 11 0F 01 01 01 00 0E 11 11 11 15 09 16 00
A490 0F 11 11 0F 05 09 11 00 0E 11 01 0E 10 11 0E 00
A4A0 1F 04 04 04 04 04 04 00 11 11 11 11 11 11 0E 00
A4B0 11 11 11 11 11 0A 04 00 11 11 11 15 15 1B 11 00
A4C0 11 11 0A 04 0A 11 11 00 11 11 0A 04 04 04 04 00
A4D0 1F 10 08 04 02 01 1F 00 1F 03 03 03 03 03 1F 00
A4E0 00 01 02 04 08 10 00 00 1F 18 18 18 18 18 1F 00
A4F0 00 00 04 0A 11 00 00 00 00 00 00 00 00 00 00 1F
A500 02 04 08 00 00 00 00 00 00 00 0E 11 1F 11 11 00
A510 00 00 0F 12 0E 12 0F 00 00 00 1E 01 01 01 1E 00
A520 00 00 0F 12 12 12 0F 00 00 00 0F 01 07 01 0F 00
A530 00 00 0F 01 07 01 01 00 00 00 1E 01 1D 11 0E 00
A540 00 00 11 11 1F 11 11 00 00 00 1F 04 04 04 1F 00
A550 00 00 0E 04 04 05 07 00 00 00 09 05 03 05 09 00
A560 00 00 01 01 01 01 1F 00 00 00 11 1B 15 11 11 00
A570 00 00 11 13 15 19 11 00 00 00 1F 11 11 11 1F 00
A580 00 00 0F 11 0F 01 01 00 00 00 1F 11 15 09 07 00
A590 00 00 1F 11 1F 05 09 00 00 00 1E 01 0E 10 0F 00
A5A0 00 00 1F 04 04 04 24 00 00 00 11 11 11 11 0E 00
A5B0 00 00 11 11 09 05 02 00 00 00 11 11 15 1B 11 00
A5C0 00 00 11 06 04 06 11 00 00 00 11 0A 04 04 04 00
A5D0 00 00 1F 08 04 02 1F 00 1C 02 04 03 04 02 1C 00
A5E0 02 04 08 10 08 04 02 00 07 08 04 18 04 08 07 00
```

```
A5F0 02 15 08 00 00 00 00 00 15 0A 15 0A 15 0A 15 00
B000 52 45 4D 4F 56 45 20 41 4E 20 45 4C 45 4D 45 4E
B010 54 20 46 52 4F 4D 20 41 20 51 55 45 55 45 20 20
B020 20 50 55 52 50 4F 53 45 3A 20 54 48 45 20 56 41
B030 52 49 41 42 4C 45 20 51 55 45 55 45 20 41 54 20
B040 4D 45 4D 4F 52 59 20 4C 4F 43 41 54 49 4F 4E 20
B050 36 30 30 30 20 43 4F 4E 54 41 49 4E 53 20 54 48
B060 45 20 41 44 52 45 53 53 20 46 4F 52 20 54 48 45
B070 20 48 45 41 44 20 4F 46 20 41 20 51 55 45 55 45
B080 2E 20 53 41 56 45 20 54 48 45 20 41 44 44 52 45
B090 53 53 20 4F 46 20 54 48 45 20 46 49 52 53 54 20
B0A0 45 4C 45 4D 45 4E 54 20 28 48 45 41 44 29 20 4F
B0B0 46 20 54 48 45 20 51 55 45 55 45 20 49 4E 20 54
B0C0 48 45 20 56 41 52 49 41 42 4C 45 20 50 4F 49 4E
B0D0 54 45 52 20 41 54 20 4D 45 4D 4F 52 59 20 4C 4F
B0E0 43 41 54 49 4F 4E 20 36 30 30 32 2E 20 55 50 44
B0F0 41 54 45 20 54 48 45 20 51 55 45 55 45 20 54 4F
B100 20 52 45 4D 4F 56 45 20 54 48 45 20 45 4C 45 4D
B110 45 4E 54 2E 20 45 41 43 48 20 45 4C 45 4D 45 4E
B120 54 20 49 4E 20 54 48 45 20 51 55 45 55 45 20 49
B130 53 20 4F 4E 45 20 57 4F 52 44 20 4C 4F 4E 47 20
B140 41 4E 44 20 43 4F 4E 54 41 49 4E 53 20 54 48 45
B150 20 41 44 44 52 45 53 53 20 4F 46 20 54 48 45 20
B160 4E 45 58 54 20 45 4C 45 4D 45 4E 54 20 49 4E 20
B170 54 48 45 20 51 55 45 55 45 2E 20 54 48 45 20 4C
B180 41 53 54 20 45 4C 45 4D 45 4E 54 20 49 4E 20 54
B190 48 45 20 51 55 45 55 45 20 43 4F 4E 54 41 49 4E
B1A0 53 20 5A 45 52 4F 20 54 4F 20 49 4E 44 49 43 41
B1B0 54 45 20 54 48 41 54 20 54 48 45 52 45 20 49 53
B1C0 20 4E 4F 20 4E 45 58 54 20 45 4C 45 4D 45 4E 54
B1D0 2E 20 51 55 45 55 45 53 20 41 52 45 20 55 53 45
B1E0 44 20 54 4F 20 53 54 4F 52 45 20 44 41 54 41 20
B1F0 49 4E 20 54 48 45 20 4F 52 44 45 52 20 49 4E 20
B200 57 48 49 43 48 20 49 54 20 57 49 4C 4C 20 42 45
B210 55 53 45 44 20 20 4F 52 20 54 41 53 4B 53 20 49
B220 4E 20 54 48 45 20 4F 52 44 45 52 20 49 4E 20 57
B230 48 49 43 48 20 54 48 45 59 20 57 49 4C 4C 20 42
B240 45 20 45 58 45 43 55 54 45 44 2E 20 54 48 45 20
B250 51 55 45 55 45 20 49 53 20 41 20 46 49 52 53 54
B260 2D 49 4E 20 46 49 52 53 54 2D 4F 55 54 20 28 46
B270 49 46 4F 29 20 44 41 54 41 20 53 54 52 55 43 54
B280 55 52 45 3B 20 54 48 41 54 20 49 53 20 20 45 4C
B290 45 4D 45 4E 54 53 20 41 52 45 20 52 45 4D 4F 56
B2A0 45 52 44 20 46 52 4F 4D 20 54 48 45 20 51 55 45
B2B0 55 45 20 49 4E 20 54 48 45 20 53 41 4D 45 20 4F
B2C0 52 44 45 52 20 49 4E 20 57 48 49 43 48 20 54 48
B2D0 45 59 20 57 45 52 45 20 45 4E 54 45 52 45 44 2E
B2E0 20 4F 50 45 52 41 54 49 4E 47 20 53 59 53 54 45
B2F0 4D 53 20 50 4C 41 43 45 20 54 41 53 4B 53 20 49
B300 4E 20 51 55 45 55 45 53 20 53 4F 20 54 48 41 54
B310 20 54 48 45 59 20 57 49 4C 4C 20 42 45 20 45 58
B320 45 43 55 54 45 44 20 49 4E 20 54 48 45 20 50 52
B330 4F 50 45 52 20 4F 52 44 45 52 2E 20 49 2F 4F 20
B340 44 52 49 56 45 52 53 20 54 52 41 4E 53 46 45 52
B350 20 44 41 54 41 20 54 4F 20 4F 52 20 46 52 4F 4D
B360 20 51 55 45 55 45 53 20 54 4F 20 45 4E 53 55 52
B370 45 20 54 48 41 54 20 54 48 45 20 44 41 54 41 20
```

……… ↓以下文章データ

```
B380 57 49 4C 4C 20 42 45 20 54 52 41 4E 53 4D 49 54
B390 54 45 44 20 4F 52 20 48 41 4E 44 4C 45 44 20 49
B3A0 4E 20 54 48 45 20 50 52 4F 50 45 52 20 4F 52 44
B3B0 45 52 2E 20 42 55 46 46 45 52 53 20 4D 41 59 20
B3C0 42 45 20 51 55 45 55 45 44 20 53 4F 20 54 48 41
B3D0 54 20 49 54 20 42 45 43 4F 4D 45 53 20 45 41 53
B3E0 59 20 54 4F 20 46 49 4E 44 20 54 48 45 20 4E 45
B3F0 58 54 20 41 56 41 49 4C 41 42 4C 45 20 42 55 46
B400 46 45 52 20 49 4E 20 41 20 53 54 4F 52 41 47 45
B410 20 50 4F 4F 4C 2E 20 51 55 45 55 45 53 20 4D 41
B420 59 20 41 4C 53 4F 20 42 45 20 55 53 45 44 20 54
B430 4F 20 4C 49 4E 4B 20 52 45 51 55 45 53 54 53 20
B440 46 4F 52 20 53 54 4F 52 41 47 45 20 20 54 49 4D
B450 49 4E 47 20 20 4F 52 20 49 2F 4F 20 54 4F 20 45
B460 4E 53 55 52 45 20 54 48 41 54 20 52 45 51 55 45
B470 53 54 53 20 41 52 45 20 53 41 54 49 53 46 49 45
B480 44 20 49 4E 20 54 48 45 20 43 4F 52 52 45 43 54
B490 20 4F 52 44 45 52 2E 20 49 4E 20 52 45 41 4C 20
B4A0 41 50 50 4C 49 43 41 54 49 4F 4E 53 20 20 20 45
B4B0 41 43 48 20 45 4C 45 4D 45 4E 54 20 49 4E 20 54
B4C0 48 45 20 51 55 45 55 45 20 57 4F 55 4C 44 20 54
B4D0 59 50 49 43 41 4C 4C 59 20 43 4F 4E 54 41 49 4E
B4E0 20 41 20 4C 41 52 47 45 20 41 4D 4F 55 4E 54 20
B4F0 4F 46 20 49 4E 46 4F 52 4D 41 54 49 4F 4E 20 41
B500 4E 44 2F 4F 52 20 53 54 4F 52 41 47 45 20 53 50
B510 41 43 45 20 49 4E 20 41 44 44 49 54 49 4F 4E 20
B520 54 4F 20 50 52 4F 56 49 44 49 4E 47 20 54 48 45
B530 20 41 44 44 52 45 53 53 20 57 48 49 43 48 20 4C
B540 49 4E 4B 53 20 45 41 43 48 20 45 4C 45 4D 45 4E
B550 54 20 54 4F 20 54 48 45 20 4E 45 58 54 20 4F 4E
B560 45 2E 20 4C 49 4E 4B 45 44 20 4C 49 53 54 53 20
B570 20 20 20 4F 4E 45 20 57 41 59 20 54 4F 20 49 4D
B580 50 4C 45 4D 45 4E 54 20 41 20 51 55 45 55 45 20
B590 49 53 20 54 4F 20 4D 41 4B 45 20 55 53 45 20 4F
B5A0 46 20 41 20 4C 49 4E 4B 45 44 20 4C 49 53 54 2E
B5B0 20 4E 4F 54 45 20 54 48 41 54 20 54 48 45 52 45
B5C0 20 49 53 20 41 20 44 49 46 46 45 52 45 4E 43 45
B5D0 20 42 45 54 57 45 45 4E 20 41 20 44 41 54 41 20
B5E0 53 54 52 55 43 54 55 52 45 20 41 4E 44 20 54 48
B5F0 45 20 49 4D 50 4C 45 4D 45 4E 54 41 54 49 4F 4E
B600 20 4F 46 20 54 48 41 54 20 44 41 54 41 20 53 54
B610 52 55 43 54 55 52 45 2E 20 46 4F 52 20 45 58 41
B620 4D 50 4C 45 20 20 41 20 51 55 45 55 45 20 49 53
B630 20 41 20 44 41 54 41 20 53 54 52 55 43 54 55 52
B640 45 20 20 41 4E 44 20 54 48 45 52 45 20 41 52 45
B650 20 4D 41 4E 59 20 44 49 46 46 45 52 45 4E 54 20
B660 57 41 59 53 20 54 48 41 54 20 59 4F 55 20 43 41
B670 4E 20 49 4D 50 4C 45 4D 45 4E 54 20 41 20 51 55
B680 45 55 45 2E 20 48 4F 57 45 56 45 52 20 20 54 48
B690 45 20 42 41 53 49 43 20 46 55 4E 43 54 49 4F 4E
B6A0 20 4F 46 20 54 48 45 20 51 55 45 55 45 20 28 46
B6B0 49 52 53 54 2D 49 4E 20 20 46 49 52 53 54 2D 4F
B6C0 55 54 29 20 49 53 20 41 4C 57 41 59 53 20 54 48
B6D0 45 20 53 41 4D 45 20 52 45 47 41 52 44 4C 45 53
B6E0 53 20 4F 46 20 54 48 45 20 57 41 59 20 49 4E 20
B6F0 57 48 49 43 48 20 59 4F 55 49 4D 50 4C 45 4D 45
B700 4E 54 20 54 48 49 53 20 44 41 54 41 20 53 54 52
```

```
B710 55 43 54 55 52 45 2E 20 20 54 48 45 20 42 41 53
B720 49 43 20 50 52 49 4E 43 49 50 4C 45 20 4F 46 20
B730 41 20 4C 49 4E 4B 45 44 20 4C 49 53 54 20 49 53
B740 20 54 48 41 54 20 45 41 43 48 20 45 4E 54 52 59
B750 20 49 4E 20 54 48 45 20 4C 49 53 54 20 43 4F 4E
B760 54 41 49 4E 53 20 54 48 45 20 41 44 44 52 45 53
B770 53 20 54 4F 20 54 48 45 20 4E 45 58 54 20 45 4E
B780 54 52 59 20 49 4E 20 54 48 45 20 4C 49 53 54 20
B790 20 49 4E 20 41 44 44 49 54 49 4F 4E 20 54 4F 20
B7A0 41 4E 59 20 44 41 54 41 20 54 48 41 54 20 4D 41
B7B0 59 20 42 45 20 46 4F 55 4E 44 20 49 4E 20 41 20
B7C0 50 41 52 54 49 43 55 4C 41 52 20 45 4C 45 4D 45
B7D0 4E 54 2E 20 4F 4E 45 20 41 44 56 41 4E 54 41 47
B7E0 45 20 4F 46 20 54 48 49 53 20 54 45 43 48 4E 49
B7F0 51 55 45 20 49 53 20 54 48 41 54 20 54 48 45 20
B800 45 4C 45 4D 45 4E 54 53 20 49 4E 20 54 48 45 20
B810 4C 49 53 54 20 44 4F 20 4E 4F 54 20 48 41 56 45
B820 20 54 4F 20 42 45 20 53 54 4F 52 45 44 20 53 45
B830 51 55 45 4E 54 49 41 4C 4C 59 20 49 4E 20 4D 45
B840 4D 4F 52 59 20 20 53 49 4E 43 45 20 45 41 43 48
B850 20 45 4E 54 52 59 20 43 4F 4E 54 41 49 4E 53 20
B860 54 48 45 20 41 44 44 52 45 53 53 20 50 4F 49 4E
B870 54 49 4E 47 20 54 4F 20 54 48 45 20 4E 45 58 54
B880 20 45 4E 54 52 59 2E 20 54 4F 20 43 48 41 4E 47
B890 45 20 54 48 45 20 4F 52 44 45 52 20 4F 46 20 54
B8A0 57 4F 20 45 4C 45 4D 45 4E 54 53 20 49 4E 20 41
B8B0 20 4C 49 4E 4B 45 44 20 4C 49 53 54 20 20 41 4C
B8C0 4C 20 59 4F 55 20 48 41 56 45 20 54 4F 20 44 4F
B8D0 20 49 53 20 4D 4F 56 45 20 50 4F 49 4E 54 45 52
B8E0 53 20 2D 20 54 48 45 20 44 41 54 41 20 41 53 53
B8F0 4F 43 49 41 54 45 44 20 57 49 54 48 20 45 41 43
B900 48 20 45 4C 45 4D 45 4E 54 20 4E 45 45 44 20 4E
B910 4F 54 20 42 45 20 4D 4F 56 45 44 2E 20 54 48 55
B920 53 20 20 54 4F 20 52 45 4D 4F 56 45 20 54 48 45
B930 20 46 49 52 53 54 20 45 4C 45 4D 45 4E 54 20 49
B940 4E 20 41 20 51 55 45 55 45 20 57 45 20 53 49 4D
B950 50 4C 59 20 4D 4F 56 45 20 41 20 43 4F 55 50 4C
B960 45 20 4F 46 20 50 4F 49 4E 54 45 52 53 20 41 4E
B970 44 20 54 48 45 20 54 41 53 4B 20 49 53 20 44 4F
B980 4E 45 3B 20 57 45 20 44 4F 4E 27 54 20 48 41 56
B990 45 20 54 4F 20 4D 4F 56 45 20 41 20 53 49 4E 47
B9A0 4C 45 20 42 49 54 20 4F 46 20 44 41 54 41 20 20
B9B0 4A 55 53 54 20 41 44 52 45 53 53 45 53 2E 20 4C
B9C0 49 4E 4B 45 44 20 4C 49 53 54 53 20 52 45 51 55
B9D0 49 52 45 20 45 58 54 52 41 20 53 54 4F 52 41 47
B9E0 45 20 41 53 20 43 4F 4D 50 41 52 45 44 20 54 4F
B9F0 20 53 45 51 55 45 4E 54 49 41 4C 20 4C 49 53 54
BA00 00 20 20 42 55 54 20 45 4C 45 4D 45 4E 54 53 20
```

3次元曲面のプログラム

グラフィックス文字描画はいかがでしたか。動かすと分かりますが，106×50文字の画面はなかなか圧巻です。

私は1文字出力ルーチンをCP/M-86の1文字出力ルーチンの仕様に合わせて，CP/M-86の画面出力を縦に表示するように改造して愛用しています。横64文字×縦80文字の表示となり，PC-9801がまるでPERQ（Three Rivers Corp. のスーパーパソコン）のようになります。ASCIIコードのコントロールコードすべてと，ダイレクトカーサ・アドレシング，画面消去などのエスケープシーケンスも組み込んだので，CP/M-86の標準ソフトウェアはもちろん，Word Masterなどのスクリーンエディタも完璧に動いています。

PC-9801でアセンブラのソースプログラムや英文の手紙を書くときに縦が24文字だと前後関係が分かりにくく，何度も前後を表示させながら書かなければなりませんが，縦に80行ないし40行もあると大変見やすいし，英文の手紙など，1画面にらくらく1枚入ってしまうのでとても重宝しています。また，DDTで連続トレースさせたり，逆アセンブルするとあっという間に画面からはみ出してしまうので，何度もやり直すことが多いものですが，縦が80行もあるとそう簡単に画面いっぱいにはなりません。

通常グラフィック画面に文字を書かせる機種(FM-8, IBM5550, PASOPIA16など）はスクロールが信じられないほど遅いのですが，PC-9801はGDCのスクロールコマンドのおかげで，瞬時にスクロールアップ/ダウン（必要ならスクロールレフト，ライトも）が可能です。もちろん，私のプログラムもスクロールコマンドを使ってスクロールさせているので，グラフィック画面に文字を書いているのにもかかわらず高速に文字が表示されます。

このプログラムは（PERQというコマンド名になっていますが）それ自身トランジェント・コマンドです。だから普通ならほかのトランジェント・コマンドと同時に使用できません。しかし，「PERQ」コマンドはTPAにロード後，自分自身をテキストVRAMに転送し，1文字出力用のジャンプベクトルを変更します。そしてテキストVRAMを見えるようにしてリブートさせます。そのため，それ以後はTPAはまったくフリーとなり，この1文字出力ルーチン

とトランジェントプログラムが競合することは絶対にありません。

ここではGDCとは直接関係ありませんが，グラフィックの例として「高速3次元グラフィック」をとり上げます。デモでよく使う帽子のような図形で，BASICで描くと数分～数十分もかかって実にいらいらするものです。

アセンブラでは「3秒弱」で図形を描画してしまいます。BASICのこの手のソフトはいくつも出ていますが，アセンブラでの速度はかなり新鮮です。

例によって，

(1) アルゴリズム

(2) BASICの例

(3) アセンブリ言語による例

で説明していきます。アセンブリ言語による例では気をつけなければならない部分，知っておくと助かること（三角関数，点の打ち方など）をくわしく説明します。

アルゴリズム

$$z=(1+\cos x)(1+\cos y)$$

の型の曲面を画面に描きます。これを原点付近で描写させると帽子のような図形になります。

この手のプログラムは，次のようにして描くのが一般的です。

(1) xを固定して，yを変化させる

(2) (x, y) に対応したzを計算

(3) 3次元の (x, y, z) に対応する画面上の点 (X, Y) を計算

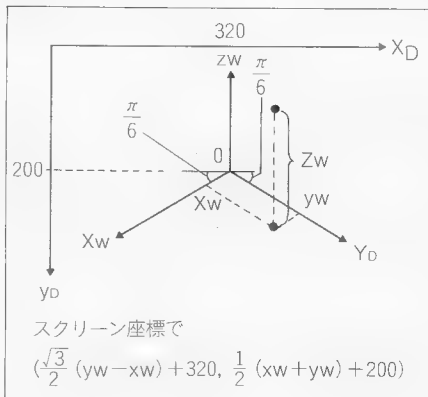
(4) yを固定してxを変化させる

(5) (2)～(4)を繰り返す

となります。この中で分かりにくいのは，(3)の「3次元の点 (x, y, z) に対応する画面上の点 (X, Y) の計算」でしょう。3次元空間では物体の位置方向，視点の位置，方向が決まって初めて「視」が決定できます。ワールド座標 (x_W, y_W, z_W) からデバイス座標 (x_D, y_D) への変換は，一般には座標交換マトリックスと透視変換マトリックスの合成で行われていますが，ここでは簡単に次式で行いました。

$$
\begin{cases}
x_D = \dfrac{\sqrt{3}}{2}(y_w - x_w) + 320 \\
y = \dfrac{1}{2}(x_w + y_w) - z_w + 200
\end{cases}
$$

この変換式は，図4-18を見ると分かりやすいでしょう。

図4-18

ほかには分かりにくいことはないでしょう。全体のフローチャートを図4-19に示します。

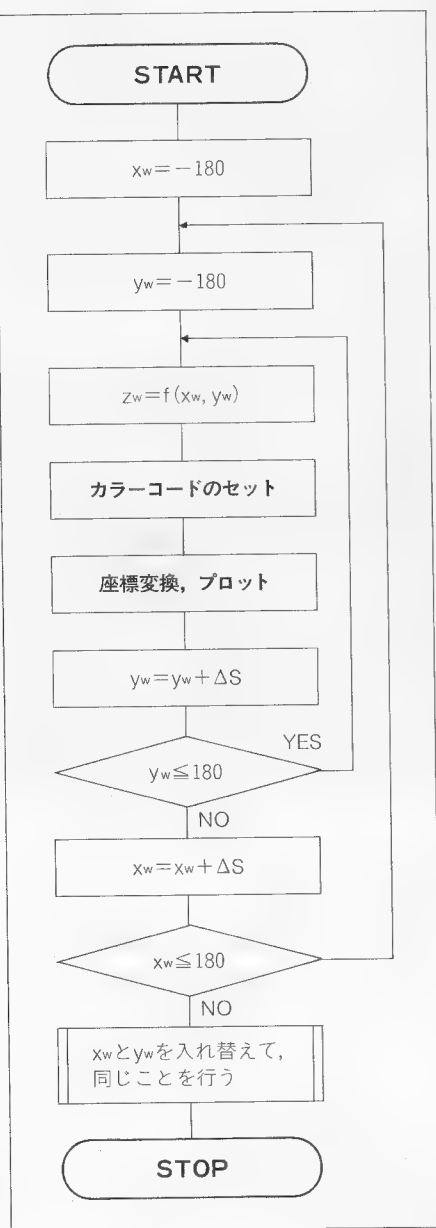
BASICにおける例

フローチャートどおりにBASICでプログラムするとリスト4-10のようになります。同じ色で表示するだけでは単調なので，高さによって色を変えています。このプログラムを動かせば，確かに

$z=(1+\cos x)(1+\cos y)$

の曲面（スケーリングしている）は描けますが，ご承知のとおり耐えられないほどの時間がかかります。BASICでどうがんばったところで，これ以上の速度は望めないので，アセンブリ言語で記述することにします。このBASICのプログラムは，いつもやっているように「アルゴリズムのチェック」の意味しかもちません。

図4-19

START
xw=−180
yw=−180
zw=f(xw, yw)
カラーコードのセット
座標変換，プロット
yw=yw+ΔS
YES
yw≦180
NO
xw=xw+ΔS
xw≦180
NO
xwとywを入れ替えて，
同じことを行う
STOP

リスト4-10 BASICによる例

```
1000 TIME$="00:00:00"
1010 SCREEN 3,1:ROLL 399:ROLL 1
1020 GOSUB *INITIALIZE
1030 FOR X.WORLD=-180 TO 180 STEP 15
1040  FOR Y.WORLD=-180 TO 180 STEP 5
1050     GOSUB *SET.Z.WORLD
1060     GOSUB *SET.Z.WORLD
1070     GOSUB *SET.X.DEVICE.Y.DEVICE
1080     GOSUB *SET.PSET.COLOR
1090     GOSUB *PLOT
1100  NEXT Y.WORLD
1110 NEXT X.WORLD
1120 FOR Y.WORLD=-180 TO 180 STEP 15
1130  FOR X.WORLD=-180 TO 180 STEP 5
1140     GOSUB *SET.Z.WORLD
1150     GOSUB *SET.Z.WORLD
1160     GOSUB *SET.X.DEVICE.Y.DEVICE
1170     GOSUB *SET.PSET.COLOR
1180     GOSUB *PLOT
1190  NEXT X.WORLD
1200 NEXT Y.WORLD
1210 PRINT TIME$
1220 END
1230 '
1240 *INITIALIZE
1250   WIDTH 80,25:CONSOLE 0,25,0,1
1260 RETURN
1270 '
1280 *SET.Z.WORLD
1290   Z.WORLD=40*(1+COS(X.WORLD/180*3.1415*11/10))*(1+COS(Y.WORLD/180*3.1415*11
/10))
1300 RETURN
1310 '
1320 *SET.X.DEVICE.Y.DEVICE
1330     X.DEVICE=320+SQR(3)/2*(Y.WORLD-X.WORLD)
1340     Y.DEVICE=200+1/2*(X.WORLD+Y.WORLD)-Z.WORLD
1350 RETURN
1360 '
1370 *SET.PSET.COLOR
1380   PSET.COLOR=INT(Z.WORLD/20) MOD 7+1
1390 RETURN
1400 '
1410 *PLOT
1420   PSET(X.DEVICE,Y.DEVICE),PSET.COLOR
1430 RETURN
```

アセンブリ言語による例

ここからがメイン部分です。

いまのアルゴリズムをアセンブリ言語で記述するには，いくつか難しいところがあります。列挙してみると，

(1) cos x の計算

(2) $(1+\cos x)(1+\cos y)$ の計算

(3) $x_D=\frac{\sqrt{3}}{2}(y_W-x_W)+320$ の計算

(4) 点を打つ：pset(x, y), color

となるでしょう。逆にいえばこれらができれば，全部をアセンブリ言語で書けるわけです。8086のインストラクションだけで作るので，内部演算は大半を符号つき整数で行っています。

①cos xの計算

「三角関数を加減乗除だけで計算しろ」といわれてすぐ思いつくのが，級数展開でしょう。つまり，

$$\begin{cases} \sin x = x - \dfrac{x^3}{3!} + \dfrac{x^5}{5!} - \dfrac{x^7}{7!} + \cdots \\ \cos x = 1 - \dfrac{x^2}{2!} + \dfrac{x^4}{4!} - \dfrac{x^6}{6!} + \cdots \end{cases}$$

です。これを適当な個数計算すれば，確かに sin x，cos x は計算できます。厳密に行うにはこの方法しかありませんが，問題は計算時間です。非常に回数の多い加減乗除算が必要なため膨大な時間がかかります。

ゲーム，特にリアルタイムに3次元図形を表示させなければならないフライトシミュレータなどでは，こんな級数展開式は絶対に使いません。では，どうやって高速に三角関数を計算しているかというと，「テーブルサーチ」という手法を使います。要するに，三角関数値を配列に入れておき，配列のインデックスで引いてくるわけです。

三角関数のような周期関数の場合は，よくこのテーブルサーチを行います。三角関数の場合は0°～90°のテーブルさえあれば，ほかの角度を求めるのは容易です。

cos x を求める必要があるので，cos x を例にして説明してみましょう。分かりやすいように，x を度（x°）で cos x° の100倍の値を返すようにします。

cos のテーブルとして，

cos-table〔0〕～cos-table〔90〕

を用意します。いうまでもなく，

$$\begin{cases} \text{cos-table}〔0〕= 100 \\ \text{cos-table}〔90〕= 0 \end{cases}$$

です。x は入力時には，$-32768 \leqq x \leqq 32767$ですが，cos x° は，360°が周期関数だから，

$\cos x^\circ = \cos(x \bmod 360)^\circ$

という関係が成り立ちます。そこで，

$x = x \bmod 360$

としても，問題ありません。

いま x は，

$-360 < x < 360$

です。さらに，

$\cos(-x^\circ) = \cos(360 - x^\circ)$

だから，

if x< 0 then x=x+360

とします。これで，x は，$0 \leqq x < 360$ となります。cos-table〔 〕は0°～90°の間の値しかもっていないので，次のようにして cos の値を求めます。

0°≦x°≦90°→cos-table〔x〕

90°<x°≦180°→-cos-table〔180-x〕

180°<x°≦270°→-cos-table〔x-180〕

270°<x°≦360°→cos-table〔360-x〕

アセンブリ言語にするときひっかかる点は，「x=x mod 360」と「テーブルを引く」ところでしょう。最初の「x=x mod 360」は「IDIV」命令で行えます。IDIV 命令は Integer Division の略で，整数除算と余りを計算してくれます。IDIV 命令は，16bit÷8bitと，32bit÷16bitの２つの種類がありますが，ここでは16ビットで演算を行っているので32bit÷16bitのほうを使います。この場合，割られる32ビットのうち，「上位16ビット」は DX レジスタに，「下位16ビット」は AX レジスタに入れ，割る数値をほかのレジスタに入れます。そして IDIV BX や IDIV CX とすれば商が AX に，余りが DX に入ってきます。

つまり，

商　　余り

DX : AX ÷ BX → AX………DX

32ビット　16ビット　16ビット　16ビット

AX に割られる数が入っていて，360で割った余りを DX に求めるには，結局，次のようにします。

```
MOV   BX, 360
CWD
IDIV  BX
```

真ん中のCWDが重要です。CWDは，Convert Word to Doudle word の略で，AXに入っている16ビットデータを，DX : AXの32ビットデータに変換します。AXの符号拡張命令です。IDIVの割られる数がDX : AXの32ビット必要だから，このコマンドは絶対に必要です。よく，IDIV BXの前にCWDを忘れることがあるので注意してください。

次にネックになるのは「テーブルサーチ」の部分でしょう。これは配列データのアクセスということになります。

「BXに配列のインデックスが入っていてラベルcos-tableからの2バイト長のデータをAXに入れる」ことを考えればよいのです。8086はアドレシングモードがかなりあるため，8ビットのCPUに比べるとプログラミングは楽です。普通1次元配列の場合，レジスタ間接モードを使います。つまり，

```
MOV   AX, 100H〔BX〕
```

という型です。これは100H番地からBX番地離れたメモリの内容を，AXにロードする命令です。

アセンブリ言語では，cos-tableがテーブルの先頭を示すラベルならば，

```
MOV AX OFFSET COS-TABLE〔BX〕
```

と書きます。「OFFSET」というのは，ラベルのオフセット値を与える「擬似命令」です。ここではワードデータでテーブルをもっているので，結局プログラムは，

```
SHL  BX, 1
MOV AX OFFSET COS-TABLE〔BX〕
```

と，BXを2倍してからテーブルを引けばよいのです。

②(1+cos x)(1+cos y)の計算

実際の計算ではスケーリングが加わっているので，

```
(100+cos(x*11/10))*(100+cos(y*11/10))/5*2/25
```

という型で計算します。

整数演算で最も気をつけなければならないのは，かけて割るのと，割ってか

けるのでは結果が違うことです。

123*100/100は123ですが

123/100*100は100となります。

あたりまえですが，よく間違える人が多いので注意してください。常に「かけてから割る」ようにしなければなりません。

もう1つ気をつけなければならないのはオーバーフローです。16ビットの整数値どうしをかけると，最大，32ビットの整数となります。これにまた16ビット数値をかけるとDX：AXの32ビットでも表せなくなります。

整数演算でプログラミングするときは，常にワーストケースを考えながら桁落ちはないか，オーバーフローはないかを注意してプログラミングしなければなりません。

IDIVを説明したので，整数乗算IMULにも少しふれておきましょう。IMULも8 bit × 8 bitと16bit × 16bitの2つの種類がありますが，ここではワード演算だから16bit × 16bitの説明をします。かけられる数は常にAXで，かける数は（AX, BX, CX, DX）です。結果は，上位16ビットがDX，下位16ビットがAXレジスタに入ります。つまり，

AX*BX → DX : AX

上位16ビット　下位16ビット

このようにDX：AXに結果がセットされるので，続いてIDIVするときに大変都合がよいのです（CWDなどの操作が不要)。

$(1+\cos x)(1+\cos y)$

の計算にはこれらの点（「かけて割る」，「オーバーフロー」）に注意してプログラミングする必要があります（リスト参照)。

③ $x_D=\frac{\sqrt{3}}{2}(y_w-x_w)+320$ の計算

この式で問題なのは$\sqrt{3}$の部分です。$\sqrt{3}$をかけるにはどうしたらよいのでしょうか。知っていればあたりまえなのですが，173をかけて100で割ればよいのです。順序を変えてはならないのは，先ほど述べたとおりです。整数演算では「かけて割る」ことが多いため，Forthなどでは*/というワードが用意されているほどです。

④点を打つ。pset(x, y), color

画面に点を打つには，要するにVRAM上の対応するアドレスに対応するデータを書き込んでやればよいのです。VRAMは何度も述べましたが，図4–20のようになっています。例として，(320, 200) に青色の点を打つことを考えてみましょう。青色プレーンだからセグメントをA800Hとします。すると，(0, 0) は0000Hとなります。x軸方向に8ドット進むごとに1アドレス増加し，y軸方向に1ドット進むごとに50Hアドレスは増加しますから，(320, 200) のアドレスは，

ADRESS＝(320￥8)＋200*50H

となります。書き込むデータは，よく考えれば分かりますが，

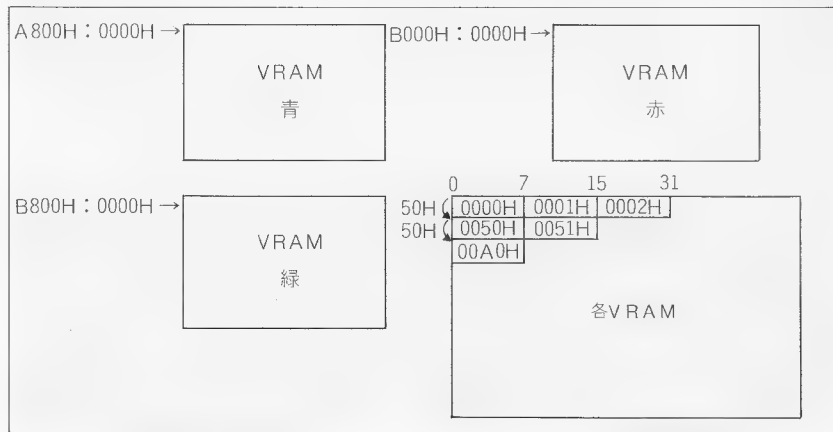
2へ (7－200 mod 8)

です。同様に，(x, y) の点の含まれるアドレス，データは

$$\begin{cases} \text{ADRESS} = x¥8 + 50H*y \\ \text{DATA} = 2^{(7-x \mod 8)} \end{cases}$$

となります。これを，VRAMに書かれているデータとORをとって書き込めば，点が打てます。

ADRESSの計算で8で割る部分がありますが，これは算術シフトを使うと高速に行えます。つまり，

図4-20

```
MOV   AX, X
SAR   AX, 1
SAR   AX, 1
SAR   AX, 1
```

とします。乗除算命令はかなり時間がかかるので，できるだけ避けたいものです。

DATAのほうは，まじめに$2^{(7-x \bmod 8)}$を計算するのではなく，論理シフトを使います。x mod 8をCLレジスタに入れて

```
MOV   AL, 128
SHR   AL, CL
```

とすればよいのです。これでALに$2^{(7-x \bmod 8)}$がセットされます。あとは，SIにADRESS，ESにA800Hを入れて

```
OR   ES : [SI], AL
```

とすれば，点が打てます。

赤の点はESをB000Hに，緑の点はESをB800Hに入れて同じことを行えばよいのです。

また，ピンクの点を打つには赤の点と青の点を打てばよいのです。

以上の点を盛り込んでプログラムしたのが，リスト4-11です。

アセンブラをもっていない人のために，ダンプリストを載せておきます（リスト4-12）。

実行時間は「3秒弱」で，あっという間にメッシュの3次元曲面を描き上げます。ぜひ動かしてください（ちなみにBASICでは5分38秒くらい）。

リスト4-11

```
;
; HIGH SPEED 3D GRAPHIC DEMONSTRATION
;
;
;

        CSEG
        ORG     0A000H
```

```
INITIALIZE:
        MOV     AX,CS
        MOV     DS,AX                   コードセグメントをデータセグメントに

        MOV     STEP2,15                大きい方のステップを15に
        MOV     STEP1,5                 (メモリにイミディエイト値を代入)
MAIN:                                   小さい方のステップを5に
        MOV     X_WORLD,-180            ワールド座標の x を－180に
MAIN1:
        MOV     Y_WORLD,-180            ワールド座標の y を－180に
MAIN2:
        CALL    SET_Z_WORLD             ワールド座標の y = f (x, y) をセット
        CALL    SET_COLOR               Z座標に応じた色にセット
        CALL    PLOT                    ワールド座標 →デバイス座標を行い点を打つ

        MOV     AX,STEP1                Xw = Yw + step 1
        ADD     Y_WORLD,AX
        CMP     Y_WORLD,180             Yw ≤ 180 ならばmain 2 へ
        JLE     MAIN2

        MOV     AX,STEP2                Xw = Xw + step 2
        ADD     X_WORLD,AX
        CMP     X_WORLD,180             Xw ≦ 180 ならばmain 1 へ
        JLE     MAIN1

MAIN3:
        MOV     Y_WORLD,-180            Yw = －180
MAIN4:
        MOV     X_WORLD,-180            Xw = －180
MAIN5:
        CALL    SET_Z_WORLD             Zwをセット
        CALL    SET_COLOR               pset _ color をセット
        CALL    PLOT                    座標変換して点を打つ

        MOV     AX,STEP1                Xw = Xw + step 1
        ADD     X_WORLD,AX
        CMP     X_WORLD,180             Yw ≦ 180 ならばmain 4 へ
        JLE     MAIN5

        MOV     AX,STEP2
        ADD     Y_WORLD,AX
        CMP     Y_WORLD,180
        JLE     MAIN4
        IRET

SET_Z_WORLD:                            Zwを計算するルーチン
:
:       INPUT (X_WORLD,Y_WORLD)
:
:       RETURN Z_=WORLD=F(X,Y)
:


        MOV     AX,X_WORLD              DX : AX = Xw • 11
        MOV     BX,11
        IMUL    BX
        MOV     BX,10                   AX = Xw • 11  10
        IDIV    BX
        CALL    COS                     cosをコールして cos(Xw • 11  10) を求める
        ADD     AX,100                  AX = 100 + cos(Xw • 11 ⁄ 10)
        SAR     AX,1
        SAR     AX,1                    AX =(100 + cos( Xw • 11 ⁄ 10)) ⁄ 4

        MOV     CX,AX                   CX =(100 + cos(Xw • 11  10))  4

        MOV     AX,Y_WORLD
```

```
        MOV     BX,11               DX : AX = Xw • 11
        IMUL    BX
        MOV     BX,10               AX : Xw • 11／10
        IDIV    BX                  AX = cos(Xw • 11／10)
        CALL    COS                 AX = 100 + cos(Xw • 11／10)
        ADD     AX,100

        IMUL    CX                  DX : AX =(100 + cos(Xw • 11／10)) • (100 + cos(Xw • 11／10))

        MOV     BX,25               AX =(100 + cos(Xw • 11／10)) • (100 + cos(Xw • 11／10))／25
        IDIV    BX
        MOV     BX,40               DX : AX =(100 + cos(Xw + 11／10)) • (100 + cos(Xw • 11／10))／25 • 40
        IMUL    BX
        MOV     BX,100              AX =(100 + cos(Xw + 11)) • (100 + cos(Xw • 11／10))／25 • 40／100
        IDIV    BX
        MOV     Z_WORLD,AX          Zw =(100 + cos(Xw • 11／10)) • (100 + cos(Yw • 11／10))／25 • 40／100
        RET                         終わり

SET_COLOR:                          色をZwに従ってセットするルーチン
;
;       SET COLOR CODE
;
;       RETURN (PSET_COLOR)
;
        MOV     AX,Z_WORLD          AX = Zw／20
        MOV     BX,20
        CWD
        IDIV    BX
        MOV     BL,7                AH =(Zw／20) mod 7
        DIV     BL
        INC     AH                  AH =(Zw／20) mod 7 + 1
        MOV     PSET_COLOR,AH       pset_color =(Zw／20) mod 7 + 1
        RET                         終わり

PLOT:                               座標を変換して点を打つルーチン
;
;       PLOT(X_WORLD,Y_WORLD,Z_WORLD)
;
;       IN GRAPHIC SCREEN
;
        MOV     AX,Y_WORLD
        SUB     AX,X_WORLD          DX : AX = 17 • (Yw − Xw)
        MOV     BX,17
        IMUL    BX
        MOV     BX,20               AX = 17 • (Yw − Xw)／20
        IDIV    BX
        ADD     AX,320              pset_X = 17／20 • (Yw − Xw) +320
        MOV     PSET_X,AX

        MOV     AX,X_WORLD
        ADD     AX,Y_WORLD          AX = Xw − Yw
        MOV     BX,2
        CWD                         AX =(Xw − Yw)／2
        IDIV    BX
        ADD     AX,200              AX =(Xw − Yw)／2 + 200
        SUB     AX,Z_WORLD          pset_Y =(Xw − Yw)／2 + 200
        MOV     PSET_Y,AX
                                    psetルーチンをコール
        CALL    PSET                終わり
        RET
                                    cosを求めるルーチン
COS:
;                                   AXの値のcosをAXに返す
;       COS ROUTINE
```

```
;
; INPUT AX ( DEGREE)                          100倍の値を返す
;
;       RETURN AX=COS(AX)*100
;
;

        MOV     BX,360
        CWD                                   DX = AX  mod  360
        IDIV    BX
        CMP     DX,0
        JGE     COS1
        ADD     DX,360                        DX ≦ 0 → DX = DX + 360
COS1:
        CMP     DX,270                        DX ≦ 270 ならばcos 2 へ
        JLE     COS2

        MOV     BX,360                        BX =(360 − DX) • 2
        SUB     BX,DX
        SHL     BX,1
        MOV     AX,OFFSET COSTABLE[BX]
        RET
COS2:
        CMP     DX,180                        DX = 180 ならばcos 3 へ
        JLE     COS3

        MOV     BX,DX                         BX =(DX − 180) • 2
        SUB     BX,180
        SHL     BX,1
        MOV     AX,OFFSET COSTABLE[BX]
        NEG     AX
        RET
COS3:
        CMP     DX,90
        JLE     COS4                          DX ≦ 90  ならばcos 4 へ

        MOV     BX,180                        BX =(180 − DX) • 2
        SUB     BX,DX
        SHL     BX,1
        MOV     AX,OFFSET COSTABLE[BX]
        NEG     AX
        RET
COS4:
        MOV     BX,DX                         BX = 2 • DX
        SHL     BX,1
        MOV     AX,OFFSET COSTABLE[BX]
        RET
PSET:                                         点を打つルーチン
;
; PSET(PSET_X,PSET_Y),PSET_COLOR
;
        TEST    PSET_COLOR,1      ! JZ    PSET1   ! CALL   PSET_BLUE
PSET1:                             pset _ color のビット 0 が立っていれば青のドットを打つ
        TEST    PSET_COLOR,2      ! JZ    PSET2   ! CALL   PSET_RED
PSET2:                             pset _ color のビット 1 が立っていれば赤のドットを打つ
        TEST    PSET_COLOR,4      ! JZ    PSET3   ! CALL   PSET_GREEN
PSET3:                             pset _ color のビット 2 が立っていれば緑のドットを打つ
        RET
PSET_BLUE:
        MOV     AX,0A800H         ! MOV    ES,AX    ES = A800H として
        CALL    PSET_ONE_PLANE                      pset _one _plane をコール
        RET
PSET_RED:                                           ES = B800H として
        MOV     AX,0B000H         ! MOV    ES,AX    pset _ one _ plane をコール
        CALL    PSET_ONE_PLANE
        RET
```

```
PSET_GREEN:
        MOV     AX,0B800H       ! MOV   ES,AX
        CALL    PSET_ONE_PLANE
        RET

PSET_ONE_PLANE:
        MOV     AX,PSET_X
        SAR     AX,1
        SAR     AX,1            SI = pset _ X/8
        SAR     AX,1
        MOV     SI,AX

        MOV     AX,PSET_Y       ! MOV   BX,80
        IMUL    BX
                                         AX = 80 * pset _ Y _ pset _ X /8

        ADD     SI,AX
        MOV     AL,128
        MOV     CX,PSET_X       ! AND   CL,7  CL = pset _ X  mod  8
                                              AL = 2 へ(7 - pset _ X  mod/8 )
        SHR     AL,CL
        OR      ES:[SI],AL        ORをとってVRAMに書き込む
        RET                       終わり


;
; COSINE TABLE
;
; 0 -> 90 ( DEGREE) *100
;
COS_TABLE:                      以下0°～90°のCOSテーブルでデータは100倍されている。
        DW      0064H,0064H,0064H,0064H,0064H,0064H,0063H,0063H,0063H,0063H
        DW      0062H,0062H,0062H,0061H,0061H,0061H,0060H,0060H,005FH,005FH
        DW      005EH,005DH,005DH,005CH,005BH,005BH,005AH,0059H,0058H,0057H
        DW      0057H,0056H,0055H,0054H,0053H,0052H,0051H,0050H,004FH,004EH
        DW      004DH,004BH,004AH,0049H,0048H,0047H,0045H,0044H,0043H,0042H
        DW      0040H,003FH,003EH,003CH,003BH,0039H,0038H,0036H,0035H,0034H
        DW      0032H,0030H,002FH,002DH,002CH,002AH,0029H,0027H,0025H,0024H
        DW      0022H,0021H,001FH,001DH,001CH,001AH,0018H,0016H,0015H,0013H
        DW      0011H,0010H,000EH,000CH,000AH,0009H,0007H,0005H,0003H,0002H
        DW      0000H

DATA    EQU     OFFSET $        DATAはコードセグメントの最後のロケーションカウンタとなる
        DSEG                    以下データセグメント
        ORG     DATA            コードセグメントのすぐ後ろにデータエリアをセット

X_WORLD DW      0
Y_WORLD DW      0
Z_WORLD DW      0

STEP2   DW      0
STEP1   DW      0               変数

PSET_X  DW      1
PSET_Y  DW      1
PSET_COLOR      DB      1

        END                     終わり
```

リスト4-12

```
A000 8C C8 8E D8 C7 06 6A A2 0F 00 C7 06 6C A2 05 00
A010 C7 06 64 A2 4C FF C7 06 66 A2 4C FF E8 58 00 E8
A020 96 00 E8 A7 00 A1 6C A2 01 06 66 A2 81 3E 66 A2
A030 B4 00 7E E8 A1 6A A2 01 06 64 A2 81 3E 64 A2 B4
A040 00 7E D3 C7 06 66 A2 4C FF C7 06 64 A2 4C FF E8
A050 25 00 E8 63 00 E8 74 00 A1 6C A2 01 06 64 A2 81
A060 3E 64 A2 B4 00 7E E8 A1 6A A2 01 06 66 A2 81 3E
A070 66 A2 B4 00 7E D3 CF A1 64 A2 BB 0B 00 F7 EB BB
A080 0A 00 F7 FB E8 77 00 05 64 00 D1 F8 D1 F8 8B C8
A090 A1 66 A2 BB 0B 00 F7 EB BB 0A 00 F7 FB E8 5E 00
A0A0 05 64 00 F7 E9 BB 19 00 F7 FB BB 28 00 F7 EB BB
A0B0 64 00 F7 FB A3 68 A2 C3 A1 68 A2 BB 14 00 99 F7
A0C0 FB B3 07 F6 F3 FE C4 88 26 72 A2 C3 A1 66 A2 2B
A0D0 06 64 A2 BB 11 00 F7 EB BB 14 00 F7 FB 05 40 01
A0E0 A3 6E A2 A1 64 A2 03 06 66 A2 BB 02 00 99 F7 FB
A0F0 05 C8 00 2B 06 68 A2 A3 70 A2 E8 53 00 C3 BB 68
A100 01 99 F7 FB 83 FA 00 7D 04 81 C2 68 01 81 FA 0E
A110 01 7E 0C BB 68 01 2B DA D1 E3 8B 87 AE A1 C3 81
A120 FA B4 00 7E 0F 8B DA 81 EB B4 00 D1 E3 8B 87 AE
A130 A1 F7 D8 C3 83 FA 5A 7E 0E BB B4 00 2B DA D1 E3
A140 8B 87 AE A1 F7 D8 C3 8B DA D1 E3 8B 87 AE A1 C3
A150 F6 06 72 A2 01 74 03 E8 15 00 F6 06 72 A2 02 74
A160 03 E8 14 00 F6 06 72 A2 04 74 03 E8 13 00 C3 B8
A170 00 A8 8E C0 E8 13 00 C3 B8 00 B0 8E C0 E8 0A 00
A180 C3 B8 00 B8 8E C0 E8 01 00 C3 A1 6E A2 D1 F8 D1
A190 F8 D1 F8 8B F0 A1 70 A2 BB 50 00 F7 EB 03 F0 B0
A1A0 80 8B 0E 6E A2 80 E1 07 D2 E8 26 08 04 C3 64 00
A1B0 63 00 63 00 63 00 63 00 63 00 63 00 63 00 63 00
A1C0 62 00 62 00 62 00 61 00 61 00 61 00 60 00 60 00
A1D0 5F 00 5F 00 5E 00 5D 00 5D 00 5C 00 5C 00 5B 00
A1E0 5A 00 59 00 59 00 58 00 57 00 56 00 55 00 54 00
A1F0 53 00 52 00 51 00 50 00 4F 00 4E 00 4D 00 4C 00
A200 4B 00 4A 00 49 00 47 00 46 00 45 00 44 00 42 00
A210 41 00 40 00 3E 00 3D 00 3C 00 3A 00 39 00 37 00
A220 36 00 34 00 33 00 31 00 30 00 2E 00 2D 00 2B 00
A230 2A 00 28 00 27 00 25 00 23 00 22 00 20 00 1E 00
A240 1D 00 1B 00 19 00 18 00 16 00 14 00 13 00 11 00
A250 0F 00 0D 00 0C 00 0A 00 08 00 06 00 05 00 03 00
A260 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
A270 00
```

入力と実行方法

MON ↵
h＝C0 ↵
としたあと，上の
ダンプを打ち込んで
CTRL－B
でBASICに
抜ける

```
SCREEN 3, 1 ↵
ROLL 399 :
  ROLL 1 ↵
DEF SEG＝0 ↵
A＝&HA000 ↵
CALL A ↵
```

で実行される

●本文注釈●

注1：少し難しくなるが，たとえばC言語で16ビット変数A, B, C, Dに対して

D=A＊B/C

という計算を有効桁16ビットで計算するには，

D=(long)A＊(long)B/(long)C

と，long すなわち32ビットデータに変換して計算させるしかない。すると，32ビットデータどうしの演算ルーチンで処理される。ところが，アセンブリ言語では，

```
MOV AX, A
IMUL B
IDIV C
MOV D, AX
```

のように，16ビットの演算だけですますことができる。32ビットデータどうしの演算は，8086の命令ではサポートされてないために，乗算は16ビットどうしの乗算を4回，加算を2回行わなければならない。除算は除算命令が使えなくなり，シフトとコンペアで書き換えなくてはならず，非常に多くの時間がかかることになる。

注2：通常の逆アセンブラでは，飛び先はアドレスで示される。そのため，逆アセンブルした結果をファイルにしても，それをアセンブラにかけることはできない。ところが，飛び先をラベルにしてファイルを作成すれば，ふたたびアセンブラにかけることもできる。このように，機械語からアセンブラにかけることのできるリストを作成できる逆アセンブラをソースジェネレータと呼ぶ。

注3：コンピュータでは，

0, 1の1つの単位をbit(ビット)	
bit 4つで	nible(ニブル)
bit 8つで	byte(バイト)
bit 16個で	word(ワード)
bit 32個で	double word(ダブル・ワード)

と呼ぶ。

注4：8088は，8086の外部データバスを8ビットにしたもので，命令はまったく8086と同一。三菱のMULTI-16，富士通のFM-11，東芝のパソピア16，日立のMB-16000等々，大半の16ビットパーソナルコンピュータは8088を採用していた。ただ，8086は一度に16ビットをフェッチできるのに対し，8088は一度に8ビットしかフェッチできないため，同一クロックを与えられた場合，8086のほうが高速になる。8086/8088の内部は，

EU (EXECUTION UNIT)

BIU (BUS INTERFACE UNIT)

に分けられるが，命令を解読するEUは8086と8088では同一なため，機械語レベルではまったく同じものといってよい。

注5：リアルタイムとは，ある入力が与えられた場合，それから一定時間以内に処理が終了することを意味する。たとえば，時計のプログラムで，1秒ごとのインタバルで割り込みがかかる場合，次のインタバル割り込みまでに，計算，表示などすべての動作が終了しなければ時計とはならない。

注6：VRAMとは (Video RAM) のことである。

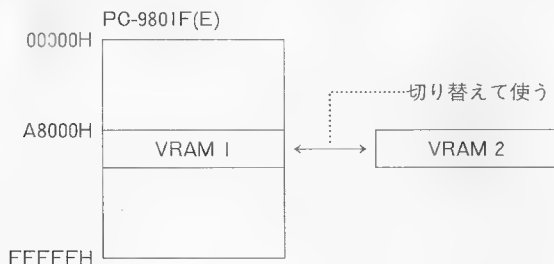
普通，ディスプレイ上に表示されるデータは，メインメモリ上に格納されるデータをディスプレイロジックが参照して表示する。つまり，画面上に文字や図形を表示させるには，メモリ中にデータを書けばよいのである。たとえば，PC-9801で，

DEF SEG＝&HB000：POKE 0，255

とメモリに書き込めば，左上に赤い横線が表示される。PC-9801のグラフィック用VRAMは1ドットに対して1ビットが対応するため，全体では

640(横)×400(縦)×3(青・赤・緑の色)

ビット，つまり96Kバイトのメモリが必要となる。

注7：バンク切り替えとは，同一メモリ空間上にいくつかのメモリを重ねてもち，それをI/O命令などで切り替えて利用する方法で，8ビットCPUなどメモリ空間の狭いCPUでよく使われる。PC-9801F(E) でもPC-9801との互換性を保ちながら，VRAMを増やすために，バンク切り替えが使われている。

注8：スタック

スタックはデータを一時蓄えるエリアのことで，実際にはメモリ上の一部が割り当てられる（ハードウェアで実現しているものもある)。スタックは最後に入れたデータが最初に出てくる (Last in First out)。これは，サブルーチンコール時の戻りアドレスの記憶や，ローカル変数，サブルーチンへの引数の一時記憶などに使われる。

注9：メモリの制約

8086の高級言語では，この制約のためにコードやデータエリアが64Kバイト以内でなければならないものも多い。

注10：エクストラセグメント

通常データセグメントはプログラムのワークエリアを指している。もしデータセグメントしか許さないとすると，ここでほかのエリア，たとえばVRAMをアクセスするにはデータセグメントレジスタを設定し直してアクセスしなければならないため，速度が落ちる。そこで，もう1つのエリア（エクストラセグメント）を許し，エクストラセグメントレジスタで指しておけば，アクセスのたびに切り替える必要はなくなる。

注11：アドレスとパラグラフ

```
0 0 0 0 0 H }
            } 0
0 0 0 1 0 H }
            } 1
0 0 0 2 0 H }
    ⋮
```

12340H }
12350H } 1234H

注12: DSの省略

「DS：に限り省略できる」という表現は厳密ではない。BPレジスタ間接のような場合は，セグメントオーバーライドプレフィックスを省略するとSS：が指定されたことになる。

省略された場合，何がデフォルトとなるかはアドレシングモードによる。

注13: ストリームキュー

8086は命令を先読みしてためておく命令ストリームキューをもっていて，実行ユニットが休まず働くようになっている。JMP命令などでアドレスが変わると，キューはリセットされ，またフェッチを始める。そのためキューがいっぱいのときが，最も実行が速くなる。

注14: フラグ

8086には演算の結果などでセット，リセットされるフラグがある。

CONTROL FLAGS
- TF……TRAP
- DF……DIRECTION
- IF ……INTERRUPT-ENABLE

STATUS FLAGS
- OF……OVERFLOW
- SF……SIGN
- ZF……ZERO
- AF……AUXILIARY CARRY
- PF……PARITY
- CF……CARRY

注15: CMP AX，BXのようなときにAX側をデスティネーション，BX側をソースという。

注16: ストリング命令は例外

MOVS …… MOVE STRING
LODS …… LOAD STRING
STOS …… STORE STRING
SCAS …… SCAN STRING
CMPS …… COMPARE STRING

注17: LOOP命令はCXレジスタを1減じ，CX≠0なら飛び CX=0ならば何もしない命令。

注18: ループ命令

LOOP命令を使えば

```
MOV   AX, 0
MOV   CX, 100
LABEL :
         ADD    AX, CX
         LOOP   LABEL
```

と簡単になる。

注19: オーバーフローフラグ

OFは算術演算による最上位ビットへのキャリーと最上位ビットからのキャリーのエクスクルーシブORをとったもの。これは符号つき2進数の加減算でオーバーフローが起きている

かどうかを示している。

注20：パリティフラグ

PFはデータ操作の結果，下位8ビットに1が偶数個あればセットされ，奇数ならばリセットされる。

注21：サインフラグ

SFは演算の最上位ビットである。
符号つきであれば，

$$最上位 = \begin{cases} 1 \rightarrow 負 \\ 0 \rightarrow 正 \end{cases}$$

となる。

注22：厳密にはMOVS命令は

メモリ→メモリ

の転送となる。

注23：BPレジスタ間接はデフォルトセグメントがSSとなることに注意。

注24：RASM86

デジタル・リサーチのリロケータブルアセンブラ。アセンブル速度はかなり速く，モジュールをリンクすることも可能。マクロ機能はない。

注25：8080モデルの指定

ここでの説明はすべて8080モデルを前提としている。そのため，GENCMDコマンドにも8080モデルであることの指定が必要。

注26：ORG

ORIGINの略

注27：DMA

DIRECT MEMORY ADDRESSの略。

注28：FCB

FILE CONTROL BLOCKの略。

注29：CCP

CONSOLE COMMAND PROCESSORの略。

注30：ラベル，変数名の長さ

もちろん，物理的な行数を超えない範囲での話である。

注31：CLI

ベクトルを書き換えている間にインタラプトがかかると，おかしなアドレスにジャンプしてしまう。これでは不都合なため，CLIでインタラプトをマスクする。

注32：B, O, D, H

BINARY, OCTAL, DECIMAL, HEXADECIMALの略。

付　録

8086 8087 オペレーションコード表

8086オペレーションコード表

ニーモニック	オペランド	オペレーションコード																バイト数	クロック数
		7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		
AAA		0	0	1	1	0	1	1	1									1	4
AAD		1	1	0	1	0	1	0	1	0	0	0	0	1	0	1	0	2	60
AAM		1	1	0	1	0	1	0	0	0	0	0	0	1	0	1	0	2	83
AAS		0	0	1	1	1	1	1	1									1	4
ADC	reg,reg	0	0	0	1	0	0	0	W	1	1	reg			r/m			2	3
	mem,reg	0	0	0	1	0	0	0	W	mod		reg			r/m			2－4	16＋EA
	reg,mem	0	0	0	1	0	0	1	W	mod		reg			r/m			2－4	9＋EA
	reg,imm	1	0	0	0	0	0	S	W	1	1	0	1	0	r/m			3－4	4
	mem,imm	1	0	0	0	0	0	S	W	mod		0	1	0	r/m			3－6	17＋EA
	acc,imm	0	0	0	1	0	1	0	W									2－3	4
ADD	reg,reg	0	0	0	0	0	0	0	W	1	1	reg			r/m			2	3
	mem,reg	0	0	0	0	0	0	0	W	mod		reg			r/m			2－4	16＋EA
	reg,mem	0	0	0	0	0	0	1	W	mod		reg			r/m			2－4	9＋EA
	reg,imm	1	0	0	0	0	0	S	W	1	1	0	0	0	r/m			3－4	4
	mem,imm	1	0	0	0	0	0	S	W	mod		0	0	0	r/m			3－6	17＋EA
	acc,imm	0	0	0	0	0	1	0	W									2－3	4
AND	reg,reg	0	0	1	0	0	0	0	W	1	1	reg			r/m			2	3
	mem,reg	0	0	1	0	0	0	0	W	mod		reg			r/m			2－4	16＋EA
	reg,mem	0	0	1	0	0	0	1	W	mod		reg			r/m			2－4	9＋EA
	reg,imm	1	0	0	0	0	0	0	W	1	1	1	0	0	r/m			3－4	4
	mem,imm	1	0	0	0	0	0	0	W	mod		1	0	0	r/m			3－6	17＋EA
	acc,imm	0	0	1	0	0	1	0	W									2－3	4
CALL	near-proc	1	1	1	0	1	0	0	0									3	19
	regptr 16	1	1	1	1	1	1	1	1	1	1	0	1	0	r/m			2	16
	memptr 16	1	1	1	1	1	1	1	1	mod		0	1	0	r/m			2－4	21＋EA
	far-proc	1	0	0	1	1	0	1	0									5	28
	memptr 32	1	1	1	1	1	1	1	1	mod		0	1	1	r/m			2－4	37＋EA

オペレーション	フラグ					
	A	C	O	P	S	Z
if((AL)&0FH)>9 or(AF)=1then(AL)←(AL)+6, (AH)←(AH)+1,(AF)←1,(CF)←(AF),(AL)←(AL)&0FH	X	X				
(AL)←(AH)＊0AH+(AL) (AH)←0	U	U	U	X	X	X
(AH)←(AL)／0AH (AL)←(AL)%0AH	U	U	U	X	X	X
if((AL)&0FH)>9 or(AF)=1 then (AL)←(AL)−6,(AH)←(AH)−1,(AF)←1 (CF)←(AF),(AL)←(AL)&0FH	×	×	U	U	U	U
(reg)←(reg)+(reg)+(CF)	×	×	×	×	×	×
(mem)←(mem)+(reg)+(CF)	×	×	×	×	×	×
(reg)←(reg)+(mem)+(CF)	×	×	×	×	×	×
(reg)←(reg)+data+(CF)	×	×	×	×	×	×
(mem)←(mem)+data+(CF)	×	×	×	×	×	×
ifW=0 AL←AL+data+(CF) ifW=1 AX←AX+data+(CF)	×	×	×	×	×	×
(reg)←(reg)+(reg)	×	×	×	×	×	×
(mem)←(mem)+(reg)	×	×	×	×	×	×
(reg)←(reg)+(mem)	×	×	×	×	×	×
(reg)←(reg)+data	×	×	×	×	×	×
(mem)←(mem)+data	×	×	×	×	×	×
ifW=0 (AL)←(AL)+data ifW=1 (AX)←(AX)+data	×	×	×	×	×	×
(reg)←(reg)&(reg)	U	0	0	×	×	×
(mem)←(mem)&(reg)	U	0	0	×	×	×
(reg)←(reg)&(mem)	U	0	0	×	×	×
(reg)←(reg)&data	U	0	0	×	×	×
(mem)←(mem)&data	U	0	0	×	×	×
ifW=0 (AL)←(AL)&data ifW=1 (AX)←(AX)&data	U	0	0	×	×	×
(SP)←(SP)−2 ((SP)+1:(SP))←(IP),(IP)←(IP)+disp						
(SP)←(SP)−2 ((SP)+1:(SP))←(IP),(IP)←(regptr 16)						
(SP)←(SP)−2 ((SP)+1:(SP))←(IP),(IP)←memptr 16						
(SP)←(SP)−2 ((SP)+1:(SP))←(CS),(CS)←seg (SP)←(SP)−2 ((SP)+1:(SP))←(IP),(IP)←offset						
(SP)←(SP)−2 ((SP)+1:(SP))←(CS),(CS)←(memptr 23+2) (SP)←(SP)−2 ((SP)+1:(SP))←(IP),(IP)←(memptr 23)						

ニーモニック	オペランド	オペレーションコード																バイト数	クロック数
		7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		
CBW		1	0	0	1	1	0	0	0									1	2
CLC		1	1	1	1	1	0	0	0									1	2
CLD		1	1	1	1	1	1	0	0									1	2
CLI		1	1	1	1	1	0	1	0									1	2
CMC		1	1	1	1	0	1	0	1									1	2
CMP	reg,reg	0	0	1	1	1	0	0	W	1	1		reg			r/m		2	3
	mem,reg	0	0	1	1	1	0	0	W	mod			reg			r/m		2 − 4	9 +EA
	reg,mem	0	0	1	1	1	0	1	W	mod			reg			r/m		2 − 4	9 +EA
	reg,imm	1	0	0	0	0	0	S	W	1	1	1	1	1		r/m		3 − 4	4
	mem,imm	1	0	0	0	0	0	S	W	mod		1	1	1		r/m		3 − 6	10+EA
	acc,imm	0	0	1	1	1	1	0	W									2 − 3	4
CMPS	dst-string, src-string	1	0	1	0	0	1	1	W									1	9 +22／ rep／22
CWD		1	0	0	1	1	0	0	1									1	5
DAA		0	0	1	0	0	1	1	1									1	4
DAS		0	0	1	0	1	1	1	1									1	4
DEC	reg8	1	1	1	1	1	1	1	0	1	1	0	0	1		r/m		2	3
	mem	1	1	1	1	1	1	1	W	mod		0	0	1		r/m		2 − 4	15+EA
	reg 16	0	1	0	0	1		reg										1	2
DIV	reg 8	1	1	1	1	0	1	1	0	1	1	1	1	0		r/m		2	80−90
	mem8	1	1	1	1	0	1	1	0	mod		1	1	0		r/m		2 − 4	(86−96) +EA

オペレーション	フラグ					
	A	C	O	P	S	Z
if(AL) < 80H then(AH)←0 else (AH)←FFH						
(CF)←0		0				
(DF)←0						
(IF)←0						
(CF)←$\overline{(CF)}$		×				
(reg)−(reg)	×	×	×	×	×	×
(mem)−(reg)	×	×	×	×	×	×
(reg)−(mem)	×	×	×	×	×	×
(reg)−data	×	×	×	×	×	×
(mem)−data	×	×	×	×	×	×
if W = 0 (AL)−data if W = 1 (AX)−data	×	×	×	×	×	×

ifW = 0 ((SI))−((DI))
if(DF) = 0 then(SI)←(SI)+1,(DI)←(DI)+1
else(SI)←(SI)−1,(DI)←(DI)−1
ifW = 1 ((SI)+1 : (SI))−((DI+1 : (DI))
if(DF) = 0 then(SI)←(SI)+2,(DI)←(DI)+2
else(SI)←(SI)−2,(DI)−2
	×	×	×	×	×	×

if(AX) < 8000H then(DX)←0 else(DX)←FFFFH						

if((AL)&0FH) > 9or(AF) = 1 then
(AL)←(AL)+6,(CF)←(AF)∨(CF),(AF)←1
if(AL) > 9FHor(CF) = 1 then
(AL)←(AL)+60H,(CF)←1
	×	×		×	×	×

if((AL)&0FH) > 9 or (AF) = 1 then
(AL)←(AL)−6,(CF)←(AF)∨(CF),(AF)←1
if(AL) > 9FHor(CF) = 1 then
(AL)←(AL)−60H,(CF)←1
	×	×	U	×	×	×

(reg8)←(reg8)−1	×		×	×	×	×
(mem)←(mem)−1	×		×	×	×	×
(reg16)←(reg16)−1	×		×	×	×	×
(temp)←(AX) if(temp)/(reg8) > FFH then (SP)←(SP)−2,((SP)+1 : (SP))←FLAGS, (IF)←0,(TF)←0 (SP)←(SP)−2,((SP)+1 : (SP))←(CS), (CS)←(2) (SP)←(SP)−2,((SP)+1 : (SP))←(IP), (IP)←(0) else(AL)←(temp)/(reg8),(AH)←(temp)%(reg8)	U	U	U	U	U	U
(temp)←(AX) if(temp)/(mem8) > FFH then (SP)←(SP)−2,((SP)+1 : (SP))←FLAGS,(IF)←0,(TF)←0 (SP)←(SP)−2,((SP)+1 : (SP))←(CS),(CS)←(2) (SP)←(SP)−2,((SP)+1 : (SP))←(IP),(IP)←(0) else(AL)←(temp)/(mem8),(AH)←(temp)%(mem8)	U	U	U	U	U	U

ニーモニック	オペランド	オペレーションコード 7 6 5 4 3 2 1 0	 7 6 5 4 3 2 1 0	バイト数	クロック数
DIV	reg 16	1 1 1 1 0 1 1 1	1 1 1 1 0 r/m	2	144－162
	mem 16	1 1 1 1 0 1 1 1	mod 1 1 0 r/m	2－4	(150－168) ＋EA
ESC	ext-op,reg	1 1 0 1 1 X X X	1 1 Y Y Y r/m	2	2
	ext-op,mem	1 1 0 1 1 X X X	mod Y Y Y r/m	2－4	8＋EA
HLT		1 1 1 1 0 1 0 0		1	2
IDIV	reg8	1 1 1 1 0 1 1 0	1 1 1 1 1 r/m	2	101－112
	mem8	1 1 1 1 0 1 1 0	mod 1 1 1 r/m	2－4	(107－118) ＋EA
	reg 16	1 1 1 1 0 1 1 1	1 1 1 1 1 r/m	2	165－184
	mem16	1 1 1 1 0 1 1 1	mod 1 1 1 r/m	2－4	(171－190) ＋EA

オペレーション	フラグ					
	A	C	O	P	S	Z
(temp)←(DX：AX) if(temp)／(reg16)>FFFFH then (SP)←(SP)−2,((SP)+1：(SP))←FLAGS,(IF)←0,(TF)←0 (SP)←(SP)−2,((SP)+1：(SP))←(CS),(CS)←(2) (SP)←(SP)−2,((SP)+1：(SP))←(IP),(IP)←(0) else(AX)←(temp)／(reg16),(DX)←(temp)%(reg16)	U	U	U	U	U	U
(temp)←(DX：AX) if(temp)／(mem16)>FFFFH then (SP)←(SP)−2,((SP)+1：(SP))←FLAGS,IF←0,TF←0 (SP)←(SP)−2,((SP)+1：(SP))←(CS),(CS)←(2) (SP)←(SP)−2,((SP)+1：(SP))←(IP),(IP)←(0) else(AX)←(temp)／(mem16),(DX)←(temp)%(mem16)	U	U	U	U	U	U
CPU escape data bus←(reg)						
CPU escape data bus←(mem)						
CPU halt						
(temp)←(AX) if(temp)／(reg8)>0and(temp)／(reg8) >7FH or(temp)／(reg8)>0 and(temp)／(reg8) <0−7FH−1 then (SP)←(SP)−2,((SP)+1：(SP))←FLAGS, (IF)←0,(TF)←0 (SP)←(SP)−2,((SP)+1：(SP))←(CS), (CS)←(2) (SP)←(SP)−2,((SP)+1：(SP))←(IP), (IP)←(0) else(AL)←(temp)／(reg8), (AH)←(temp)%(reg8)	U	U	U	U	U	U
(temp)←(AX) if(temp)／(mem8)>0 and(temp)／(mem8) >7FH or(temp)／(mem8)>0 and(temp)／mem8) <0−7FH−1 then (SP)←(SP)−2,((SP)+1：(SP))←FLAGS,(IF)←0,(TF)←0 (SP)←(SP)−2,((SP)+1：(SP))←(CS),(CS)←(2) (SP)←(SP)−2,((SP)+1：(SP))←(IP),(IP)←(0) else(AL)←(temp)／(mem8),(AH)←(temp)%(mem8)	U	U	U	U	U	U
(temp)←(DX：AX)if(temp)／(reg16)>0 and(temp)／(reg16) >7FFFH or(temp)／(reg16)<0 and(temp)／(reg16) <0−7FFFH−1 then (SP)←(SP)−2,((SP)+1：(SP))←FLAGS,(IF)←0,(TF)←0 (SP)←(SP)−2,((SP)+1：(SP))←(CS),(CS)←(2) (SP)←(SP)−2,((SP)+1：(SP))←(IP),(IP)←(0) else(AX)←(temp)／(reg16),(DX)←(temp)%(reg16)	U	U	U	U	U	U
(temp)←(DX：AX)if(temp)／(mem16)>0 and(temp)／(mem16) >7FFFH or(temp)／(mem16)<0 and(temp)／(mem16) <0−7FFFH−1 then (SP)←(SP)−2,((SP)+1：(SP))←FLAGS,IF←0,TF←0 (SP)←(SP)−2,((SP)+1：(SP))←(CS),(CS)←(2) (SP)←(SP)−2,((SP)+1：(SP))←(IP)),(IP)←(0) else(AX)←(temp)／(mem16),(DX)←(temp)%(mem16)	U	U	U	U	U	U

ニーモニック	オペランド	オペレーションコード 7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	バイト数	クロック数
IMUL	reg8	1 1 1 1 0 1 1 0	1 1 1 0 1 r/m	2	80－98
	mem8	1 1 1 1 0 1 1 0	mod 1 0 1 r/m	2－4	(86－104) ＋EA
	reg16	1 1 1 1 0 1 1 1	1 1 1 0 1 r/m	2	128－154
	mem16	1 1 1 1 0 1 1 1	mod 1 0 1 r/m	2－4	(134－160) ＋EA
IN	acc,imm8	1 1 1 0 0 1 0 W		2	10
	acc,DX	1 1 1 0 1 1 0 W		1	8
INC	reg8	1 1 1 1 1 1 1 W	1 1 0 0 0 r/m	2	3
	mem	1 1 1 1 1 1 1 W	mod 0 0 0 r/m	2－4	15＋EA
	reg16	0 1 0 0 0 reg		1	2
INT	3	1 1 0 0 1 1 0 0		1	52
	imm8 (≠3)	1 1 0 0 1 1 0 1		2	51
INTO		1 1 0 0 1 1 1 0		1	53／4
IRET		1 1 0 0 1 1 1 1		1	24
JB JNAE		0 1 1 1 0 0 1 0		2	16／4
JBE JNA		0 1 1 1 0 1 1 0		2	16 4
JCXZ		1 1 1 0 0 0 1 1		2	18 6
JE JZ	short-label	0 1 1 1 0 1 0 0		2	16／4

オ ペ レ ー シ ョ ン	フ ラ グ					
	A	C	O	P	S	Z
(AX)←(AL)＊(reg8) EXT＝AH,LOW＝AL if(EXT)＝sign extension of((LOW)then(CF)←0 else(CF)←1：(OF)←(CF)	U	×	×	U	U	U
(AX)←(AL)＊(mem8) EXT＝AH,LOW＝AL if(EXT)＝sign extention of(LOW)then(CF)←0 else(CF)←1：(OF)←(CF)	U	×	×	U	U	U
(DX：AX)＊(reg16) EXT＝DX,LOW＝AX if(EXT)＝sign extention of(LOW)then(CF)←0 else(CF)←1：(OF)←(CF)	U	×	×	U	U	U
(DX：EX)＊(mem16) EXT＝DX,LOW＝AX if(EXT)＝sign extension of(LOW)then(CF)←0 else(CF)←1：(OF)←(CF)	U	×	×	U	U	U
ifW＝0(AL)←(imm8) ifW＝1(AX)←(imm8＋1：imm8)						
ifW＝0(AL)←((DX)) ifW＝1(AX)←((DX)＋1：(DX))						
(reg8)←(reg8)＋1	×		×	×	×	×
(mem)←(mem)＋1	×		×	×	×	×
(reg16)←(reg16)＋1	×		×	×	×	×
(SP)←(SP)－2 ((SP)＋1：(SP))←FLAGS,(IF)←0,(TF)←0 (SP)←(SP)－2 ((SP)＋1：(SP))←(CS),(CS)←(14) (SP)←(SP)－2 ((SP)＋1：(SP))←(IP)),(IP)←(12)						
(SP)←(SP)－2 ((SP)＋1：(SP))←FLAGS,(IF)←0,(TF)←0 (SP)←(SP)－2 ((SP)＋1：(SP))←(CS),(CS)←(type×4＋2) (SP)←(SP)－2 ((SP)＋1：(SP))←(IP),(IP)←(type×4)						
if(OF)＝1 (SP)←(SP)－2 ((SP)＋1：(SP))←FLAGS,(IF)←0,(TF)←0 (SP)←(SP)－2 ((SP)＋1：(SP))←(CS),(CS)←(12H) (SP)←(SP)－2 ((SP)＋1：(SP))←(IP),(IP)←(10H)						
(IP)←((SP)＋1：(SP)),(SP)←(SP)＋2 (CS)←((SP)＋1：(SP)),←(SP)＋2 FLAGS←((SP)＋1：(SP)),(SP)←(SP)＋2	×	×	×	×	×	×
if(CF)＝1(IP)←(IP)＋disp						
if(CF) (ZF)＝1(IP)←(IP)＋disp						
if(CX)＝(IP)←(IP)＋disp						
if(ZF)＝1(IP)←(IP)＋disp						

ニーモニック	オペランド	オペレーションコード		バイト数	クロック数
		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0		
JL JNGE	short-label	0 1 1 1 1 1 0 0		2	16／4
JLE JNG	//	0 1 1 1 1 1 1 0		2	16／4
JMP	near-label	1 1 1 0 1 0 0 1		3	15
	short-label	1 1 1 0 1 0 1 1		2	15
	regptr16	1 1 1 1 1 1 1 1	1 1 1 0 0 r/m	2	11
	memptr16	1 1 1 1 1 1 1 1	mod 1 0 0 r/m	2 − 4	18＋EA
	far-label	1 1 1 0 1 0 1 0		5	15
	memptr32	1 1 1 1 1 1 1 1	mod 1 0 1 r/m	2 − 4	24＋EA
JNB JAE	short-label	0 1 1 1 0 0 1 1		2	16／4
JNBE JA	//	0 1 1 1 0 1 1 1		2	16／4
JNE JNZ	//	0 1 1 1 0 1 0 1		2	16／4
JNL JGE	//	0 1 1 1 1 1 0 1		2	16／4
JNLE JG	//	0 1 1 1 1 1 1 1		2	16／4
JNO	//	0 1 1 1 0 0 0 1		2	16／4
JNP JPO	//	0 1 1 1 1 0 1 1		2	16／4
JNS	//	0 1 1 1 1 0 0 1		2	16／4
JO	//	0 1 1 1 0 0 0 0		2	16／4
JP JPE	//	0 1 1 1 1 0 1 0		2	16／4
JS	//	0 1 1 1 1 0 0 0		2	16／4
LAHF		1 0 0 1 1 1 1 1		1	4
LDS	reg16, mem32	1 1 0 0 0 1 0 1	mod reg r/m	2 − 4	16＋EA
LEA	reg16, mem16	1 0 0 0 1 1 0 1	mod reg r/m	2 − 4	2＋EA
LES	reg16, mem32	1 1 0 0 0 1 0 0	mod reg r/m	2 − 4	16＋EA
LOCK		1 1 1 1 0 0 0 0		1	2
LODS	src-string	1 0 1 0 1 1 0 W		1	9＋ 13/rep /12
LOOP	short-label	1 1 1 0 0 0 1 0		2	17／5
LOOPNZ LOOPNE	//	1 1 1 0 0 0 0 0		2	19／5

オペレーション	フラグ					
	A	C	O	P	S	Z
if(SF)⊻(OF)=1 (IP)←(IP)+disp						
if[(SF)⊻(OF)]∨(ZF)=1 (IP)←(IP)+disp						
(IP)←(IP)+disp						
(IP)←(IP)+disp						
(IP)←(regptr16)						
(IP)←(memptr16)						
(CS)←seg (IP)←offset						
(CS)←(memptr+2) (IP)←(memptr32)						
if(CF)=0 (IP)←(IP)+disp						
if(CF)∨(ZF)=0 (IP)←(IP)+disp						
if(ZF)=0 (IP)←(IP)+disp						
if(SF)⊻(OF)=0 (IP)←(IP)+disp						
if[(SF)⊻(OF)]∨(ZF)=0 (IP)←(IP)+disp						
if(OF)=0 (IP)←(IP)+disp						
if(PF)=0 (IP)←(IP)+disp						
if(SF)=0 (IP)←(IP)+disp						
if(OF)=1 (IP)←(IP)+disp						
if(PF)=1 (IP)←(IP)+disp						
if(SF)=1 (IP)←(IP)+disp						
(AH)←(SF):(ZF):X:(AF):X:(PF):X:(CF)						
(reg16)←(mem32) (DS)←(mem32+2)						
(reg16) ←mem16						
(reg16)←(mem32) (ES)←(mem32+2)						
Bus Lock Prefix						

ifW=0 (AL)←((SI))
if(DF)=0 then(SI)←(SI)+1 else(SI)←(SI)−1
ifW=1 (AX)←((SI)+1:(SI))
if(DF)=0 then(SI)←(SI)+2 else(SI)←(SI)−2						

(CX)←(CX)−1 if(CX)≠0 (IP)←(IP)+disp						
(CX)←(CX)−1 if(ZF)=0 and(CX)≠0 (IP)←(IP)+disp						

ニーモニック	オペランド	オペレーションコード		バイト数	クロック数
		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0		
LOOPZ LOOPE	short-label	1 1 1 0 0 0 0 1		2	18／6
MOV	reg,reg	1 0 0 0 1 0 0 W	1 1 reg r/m	2	2
	mem,reg	1 0 0 0 1 0 0 W	mod reg r/m	2 − 4	9 +EA
	reg,mem	1 0 0 0 1 0 1 W	mod reg r/m	2 − 4	8 +EA
	mem,imm	1 1 0 0 0 1 1 W	mod 0 0 0 r/m	3 − 6	10+EA
	reg,imm	1 0 1 1 W reg		2 − 3	4
	acc,mem	1 0 1 0 0 0 0 W		3	10
	mem,acc	1 0 1 0 0 0 1 W		3	10
	sreg,reg16	1 0 0 0 1 1 1 0	1 1 0 sreg r/m	2	2
	sreg,mem	1 0 0 0 1 1 1 0	mod 0 sreg r/m	2 − 4	8 +EA
	reg16,sreg	1 0 0 0 1 1 0 0	1 1 0 sreg r/m	2	2
	mem,sreg	1 0 0 0 1 1 0 0	mod 0 sreg r/m	2 − 4	9 +EA
MOVS MOVSB * MOVSW *	dst-string, src-string	1 0 1 0 0 1 0 W		1	9 + 17/rep /18
MUL	reg8	1 1 1 1 0 1 1 0	1 1 1 0 0 r/m	2	70−77
	mem8	1 1 1 1 0 1 1 0	mod 1 0 0 r/m	2 − 4	(76−83) +EA
	reg16	1 1 1 1 0 1 1 1	1 1 1 0 0 r/m	2	118−133
	mem16	1 1 1 1 0 1 1 1	mod 1 0 0 r/m	2 − 4	(124−139) +EA
NEG	reg	1 1 1 1 0 1 1 W	1 1 0 1 1 r/m	2	3
	mem	1 1 1 1 0 1 1 W	mod 0 1 1 r/m	2 − 4	16+EA
NOP		1 0 0 1 0 0 0 0		1	3
NOT	reg	1 1 1 1 0 1 1 W	1 1 0 1 0 r/m	2	3
	mem	1 1 1 1 0 1 1 W	mod 0 1 0 r/m	2 − 4	16+EA
OR	reg,reg	0 0 0 0 1 0 0 W	1 1 reg r/m	2	3
	mem,reg	0 0 0 0 1 0 0 W	mod reg r/m	2 − 4	16+EA
	reg,mem	0 0 0 0 1 0 1 W	mod reg r/m	2 − 4	9 +EA
	reg,imm	1 0 0 0 0 0 0 W	1 1 0 0 1 r/m	3 − 4	4
	mem,imm	1 0 0 0 0 0 0 W	mod 0 0 1 r/m	3 − 6	17+EA
	acc,imm	0 0 0 0 1 1 0 W		2 − 3	4
OUT	imm8,acc	1 1 1 0 0 1 1 W		2	10

*：オペランドなし

オペレーション	フラグ					
	A	C	O	P	S	Z
(CX)←(CX)−1 if(ZF)=1 and(CX)≠0 (IP)←(IP)+disp						
(reg)←(reg)						
(mem)←(reg)						
(reg)←(mem)						
(mem)←(data)						
(reg)←(data)						
ifW=0 (AL)←(addr) ifW=1 (AX)←(addr+1 : addr)						
ifW=0 (addr)←(AL) ifW=1 (addr+1 : addr)←(AX)						
(sreg)←(reg16)						
(sreg)←(mem)						
(reg16)←(sreg)						
(mem)←(sreg)						

ifW=0 ((DI))←((SI))
if(DF)=0 then (SI)←(SI)+1,(DI)←(DI)+1
else(SI)←(SI)−1,(DI)→(DI)−1
ifW=1 ((DI)+1 : (DI))←((SI)+1 : (SI))
if(DF)=0 then(SI)←(SI)+2,(DI)←(DI)+2
else(SI)←(SI)−2,(DI)←(DI)−2						

(AX)←(AL)∗(reg8) EXT=AH if(EXT)=0 then(CF)←0 else(CF)←1 : (OF)←(CF)	U	×	×	U	U	U
(AX)←(AL)∗(mem8) EXT=AH if(EXT)=0 then(CF)←0 else(CF)←1 : (OF)←(CF)	U	×	×	U	U	U
(DX : AX)←(AX)∗(reg16) EXT=DX if(EXT)=0 then(CF)←0 else (CF)←1 : (OF)←(CF)	U	×	×	U	U	U
(DX : AX)←(AX)∗(mem16) EXT=DX if(EXT)=0 then(CF)←0 else(CF)←1 : (OF)←(CF)	U	×	×	U	U	U
(reg)←0−(reg)	×	×	×	×	×	×
(mem)←0−(mem)	×	×	×	×	×	×
ノー・オペレーション						
ifW=0 (reg8)←FFH−(reg8) ifW=1 (reg16)←FFFFH−(reg16)						
ifW=0 (mem8)←FFH−(mem8) ifW=1 (mem16)←FFFFH−(mem16)						
(reg)←(reg)∨(reg)	U	0	0	×	×	×
(mem)←(mem)∨(reg)	U	0	0	×	×	×
(reg)←(reg)∨(mem)	U	0	0	×	×	×
(reg)←(reg) data	U	0	0	×	×	×
(mem)←(mem)∨data	U	0	0	×	×	×
ifW=0 (AL)←(AL)∨data ifW=1 (AX)←(AX)∨data	U	0	0	×	×	×
ifW=0 (imm8)←(AL) ifW=1 (imm8+1 : imm8)←(AX)						

ニーモニック	オペランド	オペレーションコード 7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	バイト数	クロック数
OUT	DX,acc	1 1 0 1 1 1 W W		1	8
POP	mem	1 0 0 0 1 1 1 1	mod 0 0 0 r/m	2 − 4	17+EA
	reg	0 1 0 1 1 reg		1	8
	sreg	0 0 0 sreg 1 1 1		1	8
POPF		1 0 0 1 1 1 0 1		1	8
PUSH	mem	1 1 1 1 1 1 1 1	mod 1 1 0 r/m	2 − 4	16+EA
	reg	0 1 0 1 0 reg		1	10
	sreg	0 0 0 sreg 1 1 0		1	10
PUSHF		1 0 0 1 1 1 0 0		1	10
RCL	reg,1	1 1 0 1 0 0 0 W	1 1 0 1 0 r/m	2	2
	mem,1	1 1 0 1 0 0 0 W	mod 0 1 0 r/m	2 − 4	15+EA
	reg,CL	1 1 0 1 0 0 1 W	1 1 0 1 0 r/m	2	8 + 4/bit
	mem,CL	1 1 0 1 0 0 1 W	mod 0 1 0 r/m	2 − 4	20+EA +4/bit
RCR	reg,1	1 1 0 1 0 0 0 W	1 1 0 1 1 r/m	2	2
	mem,1	1 1 0 1 0 0 0 W	mod 0 1 1 r/m	2 − 4	15+EA
	reg,CL	1 1 0 1 0 0 1 W	1 1 0 1 1 r/m	2	8 + +4/bit

オペレーション	フラグ					
	A	C	O	P	S	Z
ifW＝0 ((DX))←(AL) ifW＝1 ((DX)＋1：(DX))←(AX)						
(mem)←((SP)＋1：(SP)) (SP)←(SP)＋2						
(reg)←((SP)＋1：(SP)) (SP)←(SP)＋2						
(sreg)←((SP)＋1：(SP)) (SP)←(SP)＋2						
FLAGS←((SP)＋1：(SP)) (SP)←(SP)＋2	R	R	R	R	R	R
(SP)←(SP)－2 ((SP)＋1：(SP))←(mem)						
(SP)←(SP)－2 ((SP)＋1：(SP))←(reg)						
(SP)←(SP)－2 ((SP)＋1：(SP))←(sreg)						
(SP)←(SP)－2 ((SP)＋1：(SP))←FLAGS						
(tmpcf)←(CF),(CF)←high-order bit of(reg) (reg)←(reg)＊2＋(tmpcf) if high-order bit of(reg)←(CF) then(OF)←1 else(OF)←0		×	×			
(tmpcf)←(CF),(CF)←high-order bit of(mem) (mem)←(mem)＊2＋(tmpct) if high-order bit of(mem)≠(CF) then(OF)←1 else(OF)←0		×	×			
(temp)←(CL)do while(temp)≠0 (tmpcf)←(CF),(CF)←high-order bit of(reg) (reg)←(reg)＊＋2(tmpct) (temp)←(temp)－1 (OF)undefined		×	U			
(temp)←(CL) do while(temp)≠0 (tmpct)←(CF),(CF)←high-order bit of(mem) (mem)←(mem)＊2＋(tmpcf) (temp)←(temp)－1 (OF)undefined		×	U			
(tempcf)←(CF), (CF)←low-order bit of(reg),(reg)←(reg)／2 high-order bit of(reg)←(tempcf) if high-order bit of(reg)≠next-to-high-order bit of(reg) then(OF)←1 else(OF)←0		×	×			
(tmpcf)←(CF), (CF)←low-order bit of(mem),(mem)←(mem)／2 high-order bit of(mem)←(tmpcf) if high-order bit of(mem)≠next-to-high-order bit of mem(OF)←1 else(OF)←0		×	×			
(temp)←(CL) do while (tmp)≠0 (tmpcf)←(CF),(CF)←low-order bit of(reg) (reg)←(reg)／2 high－order bit of(r/m)←(tmpcf) (temp)←(temp)－1 (OF)undefined		×	U			

ニーモニック	オペランド	オペレーションコード 7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	バイト数	クロック数
RCR	mem,CL	1 1 0 1 0 0 1 W	mod 0 1 1 r/m	2 − 4	20＋EA ＋4/bit
REP REPE REPZ		1 1 1 1 0 0 1 1		1	2
REPNE REPNZ		1 1 1 1 0 0 1 0		1	2
RET		1 1 0 0 0 0 1 1		1	8
	pop-value	1 1 0 0 0 0 1 0		3	12
		1 1 0 0 1 0 1 1		1	18
	pop-value	1 1 0 0 1 0 1 0		3	17
ROL	reg,1	1 1 0 1 0 0 0 W	1 1 0 0 0 r/m	2	2
	mem,1	1 1 0 1 0 0 0 W	mod 0 0 0 r/m	2 − 4	15＋EA
	reg,CL	1 1 0 1 0 0 1 W	1 1 0 0 0 r/m	2	8 ＋ 4/bit
	mem,CL	1 1 0 1 0 0 1 W	mod 0 0 0 r/m	2 − 4	20＋EA ＋4/bit
ROR	reg,1	1 1 0 1 0 0 0 W	1 1 0 0 1 r/m	2	2
	mem,1	1 1 0 1 0 0 0 W	mod 0 0 1 r/m	2 − 4	15＋EA

オペレーション	フラグ					
	A	C	O	P	S	Z
(temp)←(CL) do while (temp)≠0 (tmpcf)←(CF),(CF)←low-order bit of(mem) (mem)←(mem)／2 high-order bit of(mem)←(tmpcf) (temp)←(temp)－1 (OF)undefined		×	U			
do while (CX)≠0 続くバイトのプリミティブ・ストリング命令を実行する (CX)←(CX)－1 保留割り込みがあれば処理する プリミティブ命令がCMPSまたはSCASでかつ，(ZF)≠1のときループを抜け出る。						
do while (CX)≠0 続くバイトのプリミティブ・ストリング命令を実行する (CX)←(CX)－1 保留割り込みがあれば処理する (ZF)≠0のときループを抜け出る						
(IP)←((SP)＋1：(SP)) (SP)←(SP)＋2						
(IP)←((SP)＋1：(SP)) (SP)←(SP)＋2,(SP)←(SP)＋data						
(IP)←((SP)＋1：(SP)) (SP)←(SP)＋2 (CS)←((SP)＋1：(SP)) (SP)←(SP)＋2						
(IP)←((SP)＋1：(SP)) (SP)←(SP)＋2 (CS)←((SP)＋1：(SP)) (SP)←(SP)＋2,(SP)←(SP)＋data						
(CF)←high-order bit of(reg),(reg)←(reg)＊2＋(CF) if high-order bit of(reg)≠(CF) then(OF)←1 else(OF)←0		×	×			
(CF)←high-order bit of(mem),(mem)←(mem)＊2＋(CF) if high-order bit of(mem)≠(CF) then(OF)←1 else(OF)←0		×	×			
(temp)←(CL) do while(temp)≠0 (CF)←high-order bit of(reg),(reg)＊2＋(CF) (temp)←(temp)－1 (OF)undefined		×	U			
(temp)←(CL) do while(temp)≠0 (CF)←high-order bit of(mem),(mem)←(mem)＊2＋(CF) (temp)←(temp)－1 (OF)undefined		×	U			
(CF)←low-order bit of(reg),(reg)←(reg)／2 high-order bit of(reg)←(CF) if high-order bit of(reg)≠next-to-high-order bit of(reg) then(OF)←1 else(OF)←0		×	×			
(CF)←low-order bit of(mem),(mem)←(mem)／2 high-order bit of(mem)←(CF) if high-order bit of(mem)≠next-to-high-order bit of(mem)then(OF)←1 else(OF)←0		×	×			

ニーモニック	オペランド	オペレーションコード		バイト数	クロック数
		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0		
ROR	reg,CL	1 1 0 1 0 0 1 W	1 1 0 0 1 r/m	2	8+4/bit
	mem,CL	1 1 0 1 0 0 1 W	mod 0 0 1 r/m	2 − 4	20+EA +4/bit
SAHF		1 0 0 1 1 1 1 0		1	4
SAR	reg,1	1 1 0 1 0 0 0 W	1 1 1 1 1 r/m	2	2
	mem,1	1 1 0 1 0 0 0 W	mod 1 1 1 r/m	2 − 4	15+EA
	reg,CL	1 1 0 1 0 0 1 W	1 1 1 1 1 r/m	2	8 + 4/bit
	mem,CL	1 1 0 1 0 0 1 W	mod 1 1 1 r/m	2 − 4	20+EA +4/bit
SBB	reg,reg	0 0 0 1 1 0 0 W	1 1 reg r/m	2	3
	mem,reg	0 0 0 1 1 0 0 W	mod reg r/m	2 − 4	16+EA
	reg,mem	0 0 0 1 1 0 1 W	mod reg r/m	2 − 4	9 +EA
	reg,imm	1 0 0 0 0 0 S W	1 1 0 1 1 r/m	3 − 4	4
	mem,imm	1 0 0 0 0 0 S W	mod 0 1 1 r/m	3 − 6	17+EA
	acc,imm	0 0 0 1 1 1 0 W		2 − 3	4
SCAS	dst-string	1 0 1 0 1 1 1 W		1	9 + 15/rep /15
SHL SAL	reg,1	1 1 0 1 0 0 0 W	1 1 1 0 0 r/m	2	2
	mem,1	1 1 0 1 0 0 0 W	mod 1 0 0 r/m	2 − 4	15+EA
	reg,CL	1 1 0 1 0 0 1 W	1 1 1 0 0 r/m	2	8 + 4/bit
	mem,CL	1 1 0 1 0 0 1 W	mod 1 0 0 r/m	2 − 4	20+EA +4/bit
SHR	reg,1	1 1 0 1 0 0 0 W	1 1 1 0 1 r/m	2	2
	mem,1	1 1 0 1 0 0 0 W	mod 1 0 1 r/m	2 − 4	15+EA

オペレーション	フラグ					
	A	C	O	P	S	Z
(temp)←(CL) do while(temp)≠0 (CF)←low-order bit of(reg),(reg)←(reg)/2 high-order bit of(reg)←(CF) (temp)←(temp)−1 (OF)undefined		X	U			
(temp)←(CL) do while(temp)←0 (CF)←low-order bit of(mem),(mem)←(mem)/2 high-order bit of(mem)←(CF) (temp)←(temp)−1 (OF)undefined		X	U			
(SF):(ZF):X:(AF):X:(PF):X:(CF)←(AH)	X	X		X	X	X
(CF)←low-order bit of(reg),(reg)←(reg)/2,(OF)←0	U	X	0	X	X	X
(CF)←low-order bit of(mem),(mem)←(mem)/2,(OF)←0	U	X	0	X	X	X
(temp)←(CL) do while(temp)≠0 (CF)←low-order bit of(reg),(reg)←(reg)/2 (temp)←(temp)−1 (OF)undefined	U	X	U	X	X	X
(temp)←(CL) do while(temp)≠0 (CF)←low-order bit of(mem),(mem)←(mem)/2 (temp)←(temp)−1 (OF)undefined	U	X	U	X	X	X
(reg)←(reg)−(reg)−(CF)	×	×	×	×	×	×
(mem)←(mem)−(reg)−(CF)	×	×	×	×	×	×
(reg)←(reg)−(mem)−(CF)	×	×	×	×	×	×
(reg)←(reg)−data−(CF)	×	×	×	×	×	×
(mem)←(mem)−data−(CF)	×	×	×	×	×	×
ifW=0 (AL)←(AL)−data−(CF) ifW=1 (AX)←(AX)−data−(CF)	×	×	×	×	×	×

ifW=0 (AL)−((DI))
if(DF)=0 then(DI)←(DI)+1 else(DI)←(DI)−1
ifW=1 (AX)−((DI)+1:(DI))
if(DF)=0 then(DI)←(DI)+2 else(DI)←(DI)−2
	×	×	×	×	×	×

(CF)←high-order bit of(reg),(r/m)←(reg)*2 if high-order bit of(reg)≠(CF) then(OF)←1 else(OF)←0	U	×	×	×	×	×
(CF)←high-order bit of(mem),(mem)←(mem)*2 if high-order bit of (EA)≠(CF)then(OF)←1 else(OF)←0	U	×	×	×	×	×
(temp)←(CL), do while(temp)≠0 (CF)←high-order bit of(reg),(reg)←(reg)*2 (temp)←(temp)−1 (OF)undefined	U	×	U	×	×	×
(temp)←(CL),do while(temp)≠0 (CF)←high-order bit of(mem),(mem)←(mem)*2 (temp)←(temp)−1 (OF)undefined	U	×	U	×	×	×
(CF)←low-order bit of(reg),(reg)←(reg)/2 if high-order bit of(reg)≠next-to-high-order bit of(reg)then(OF)←1 else(OF)←0	U	×	×	×	×	×
(CF)←low-order bit of(mem),(mem)←(mem)/2 if high-order bit of(mem)≠next-to-high-order bit of(mem)then(OF)←1 else(OF)←0	U	×	×	×	×	×

ニーモニック	オペランド	オペレーションコード		バイト数	クロック数
		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0		
SHR	reg,CL	1 1 0 1 0 0 1 W	1 1 1 0 1 r/m	2	8 + 4/bit
	mem,CL	1 1 0 1 0 0 1 W	mod 1 0 1 r/m	2 − 4	20+EA +4/bit
STC		1 1 1 1 1 0 0 1		1	2
STD		1 1 1 1 1 1 0 1		1	2
STI		1 1 1 1 1 0 1 1		1	2
STOS	dst-string	1 0 1 0 1 0 1 W		1	9 + 10/rep /11
SUB	reg,reg	0 0 1 0 1 0 0 W	1 1 reg r/m	2	3
	mem,reg	0 0 1 0 1 0 0 W	mod reg r/m	2 − 4	16+EA
	reg,mem	0 0 1 0 1 0 1 W	mod reg r/m	2 − 4	9 +EA
	reg,imm	1 0 0 0 0 0 S W	1 1 1 0 1 r/m	3 − 4	4
	mem,imm	1 0 0 0 0 0 S W	mod 1 0 1 r/m	3 − 6	17+EA
	acc,imm	0 0 1 0 1 1 0 W		2 − 3	4
TEST	reg,reg	1 0 0 0 0 1 0 W	1 1 reg r/m	2	3
	mem,reg	1 0 0 0 0 1 0 W	mod reg r/m	2 − 4	9 +EA
	reg,imm	1 1 1 1 0 1 1 W	1 1 0 0 0 r/m	3 − 4	5
	mem,imm	1 1 1 1 0 1 1 W	mod 0 0 0 r/m	3 − 6	11+EA
	acc,imm	1 0 1 0 1 0 0 W		2 − 3	4
WAIT		1 0 0 1 1 0 1 1		1	3+5n
XCHG	reg,reg	1 0 0 0 0 1 1 W	1 1 reg r/m	2	4
	mem,reg	1 0 0 0 0 1 1 W	mod reg r/m	2 − 4	17+EA
	AX,reg16	1 0 0 1 0 reg		1	3
XLAT	source -table	1 1 0 1 0 1 1 1		1	11
XOR	reg,reg	0 0 1 1 0 0 0 W	1 1 reg r/m	2	3
	mem,reg	0 0 1 1 0 0 0 W	mod reg r/m	2 − 4	16+EA
	reg,mem	0 0 1 1 0 0 1 W	mod reg r/m	2 − 4	9 +EA
	reg,imm	1 0 0 0 0 0 0 W	1 1 1 1 0 r/m	3 − 4	4
	mem,imm	1 0 0 0 0 0 0 W	mod 1 1 0 r/m	3 − 6	17+EA
	acc,imm	0 0 1 1 0 1 0 W		2 − 3	4

(注)

EA＝
- ダイレクトモード　6
- レジスタ間接モード　5
- ベースorインデクスモード　9
- ベースインデクスモード(ディスプレースメントなし)　7 or 8
- ベースインデクスモード(ディスプレースメント付き)　11 or 12

オペレーション	フラグ					
	A	C	O	P	S	Z
(temp)←(CL) do while(temp)≠0 (CF)←low-order bit of(reg),(reg)←(reg)／2 (temp)←(temp)－1 (OF)undefined	U	×	U	×	×	×
(temp)←(CL) do while(temp)≠0 (CF)←low-order bit of(mem),(mem)←(mem)／2 (temp)←(temp)－1 (OF)undefined	U	×	U	×	×	×
(CF)←1		1				
(DF)←1						
(IF)←1						

ifW＝0 ((DI))←(AL)
if(DF)＝0 then(DI)←(DI)＋1 else(DI)←(DI)－1
ifW＝1 ((DI)＋1：(DI))←(AX)
if(DF)＝0 then(DI)←(DI)＋2 else(DI)←(DI)－2						

(reg)←(reg)－(reg)	×	×	×	×	×	×
(mem)←(mem)－(reg)	×	×	×	×	×	×
(reg)←(reg)－(mem)	×	×	×	×	×	×
(reg)←(reg)－data	×	×	×	×	×	×
(mem)←(mem)－data	×	×	×	×	×	×
ifW＝0 (AL)←(AL)－data ifW＝1 (AX)←(AX)－data	×	×	×	×	×	×
(reg)&(reg)	U	0	0	×	×	×
(reg)&(mem)	U	0	0	×	×	×
(reg)& data	U	0	0	×	×	×
(mem)& data	U	0	0	×	×	×
ifW＝0 (AL)& data ifW＝1 (AX)& data	U	0	0	×	×	×
CPU wait						
(reg)←→(reg)						
(reg)←→(mem)						
(reg16)←→(AX)						
(AL)←((BX)＋(AL))						
(reg)←(reg)⩛(reg)	U	0	0	×	×	×
(mem)←(mem)⩛(reg)	U	0	0	×	×	×
(reg)←(reg)⩛(mem)	U	0	0	×	×	×
(reg)←(reg)⩛data	U	0	0	×	×	×
(mem)←(mem)⩛data	U	0	0	×	×	×
ifW＝0 (AL)←(AL)⩛data ifW＝1 (AX)←(AX)⩛data	U	0	0	×	×	×

8087オペレーションコード表

ニーモニック	オペランド	オペレーションコード																バイト数
		7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
FABS	—	ESCAPE					0	0	1	1	1	1	0	0	0	0	1	2
FADD	// ST,ST(i)/ST(i),ST	ESCAPE					d	0	0	1	1	0	0	0	ST(i)			2
	short-real	ESPACE					0	0	0	MOD		0	0	0	R/M			2−4
	long-real	ESCAPE					1	0	0	MOD		0	0	0	R/M			2−4
FADDP	ST(i),ST	ESCAPE					d	1	0	1	1	0	0	0	ST(i)			2
FBLD	packed-decimal	ESCAPE					1	1	1	MOD		1	0	0	R/M			2−4
FBSTP	packed-decimal	ESCAPE					1	1	1	MOD		1	1	0	R/M			2−4
FCHS	—	ESCAPE					0	0	1	1	1	1	0	0	0	0	0	2
FCLEX/ FNCLEX	—	ESPACE					0	1	1	1	1	1	0	0	0	1	0	2
FCOM	// ST(i)	ESCAPE					0	0	0	1	1	0	1	0	ST(i)			2
	short-real	ESCAPE					0	0	0	MOD		0	1	0	R/M			2−4
	long-real	ESCAPE					1	0	0	MOD		0	1	0	R/M			2−4
FCOMP	// ST(i)	ESCAPE					0	0	0	1	1	0	1	1	ST(i)			2
	short-real	ESCAPE					0	0	0	MOD		0	1	1	R/M			2−4
	long-real	ESCAPE					1	0	0	MOD		0	1	1	R/M			2−4
FCOMPP	—	ESCAPE					1	1	0	1	1	0	1	1	0	0	1	2
FDECSTP	—	ESCAPE					0	0	1	1	1	1	1	0	1	1	0	2
FDISI/ FNDISI	—	ESCAPE					0	1	1	1	1	1	0	0	0	0	1	2
FDIV	// ST(i),ST	ESCAPE					d	0	0	1	1	1	1	0	R/M			2
	short-real	ESCAPE					0	0	0	MOD		1	1	0	R/M			2−4
	long-real	ESCAPE					1	0	0	MOD		1	1	0	R/M			2−4
FDIVP	ST(i),ST	ESCAPE					d	1	0	1	1	1	1	0	R/M			2
FDIVR	// ST,ST(i)/ST(i),ST	ESCAPE					d	0	0	1	1	1	1	1	R/M			2
	short-real	ESCAPE					0	0	0	MOD		1	1	1	R/M			2−4
	long-real	ESCAPE					1	0	0	MOD		1	1	1	R/M			2−4
FDIVRP	ST(i),ST	ESCAPE					d	1	0	1	1	1	1	1	R/M			2
FENI/ FNENI	—	ESCAPE					0	1	1	1	1	1	0	0	0	0	0	2
FFREE	ST(i)	ESCAPE					1	0	1	1	1	0	0	0	ST(i)			2
FIADD	word-integer	ESCAPE					1	1	0	MOD		1	0	0	R/M			2−4
	short-integer	ESCAPE					0	1	0	MOD		0	0	0	R/M			2−4
FICOM	word-integer	ESCAPE					1	1	0	MOD		0	1	0	R/M			2−4
	short-integer	ESCAPE					0	1	0	MOD		0	1	0	R/M			2−4
FICOMP	word-integer	ESCAPE					1	1	0	MOD		0	1	1	R/M			2−4
	short-integer	ESCAPE					0	1	0	MOD		0	1	1	R/M			2−4
FIDIV	word-integer	ESCAPE					1	1	0	MOD		1	1	0	R/M			2−4
	short-integer	ESCAPE					0	1	0	MOD		1	1	0	R/M			2−4

クロック数		転送		コーディング例	コメント
Typical	Range	8086	8088		
14	10−17	0	0	FABS	絶対比
85 105+EA 110+EA	70−100 90−120+EA 95−125+EA	0 2/4 4/6	0 4 8	FADD ST,ST(4) FADD AIR_TEMP〔SI〕 FADD〔BX〕MEAN	実数加算
90	75−105	0	0	FADDP ST(2),ST	フローティング加算とポップ
300+EA	290−310+EA	5/7	10	FBLD YTD_SALES	BCDロード
530+EA	520−540+EA	6/8	12	FBSTP〔BX〕.FORECAST	BCDストアとポップ
15	10−17	0	0	FCHS	符号反転
5	2 − 8	0	0	FNCLEX	クリアエクセプション
45 65+EA 70+EA	40−50 60−70+EA 65−75+EA	0 2/4 4/6	0 4 8	FCOM ST(1) FCOM 〔BP〕.UPPER_LIMIT FCOM WAVELENGTH	実数比較
47 68+EA 72+EA	42−52 63−73+EA 67−77+EA	0 2/4 2/6	0 4 8	FCOMP ST(2) FCOMP (BF+2).N_READINGS FCOMP DENSITY	実数比較とポップ
50	45−55	0	0	FCOMPP	実数比較と2回のポップ
9	6 −12	0	0	FDECSTP	デクリメントスタックポインタ
5	2 − 8	0	0	FDISI	インタラプト禁止
198 220+EA 225+EA	193−203 215−225+EA 220−230+EA	0 2/4 4/6	0 4 8	FDIV FDIV DISTANCE FDIV ARC〔DI〕	実数除算
202	197−207	0	0	FDIVP ST(4),ST	実数除算とポップ
199 221+EA 226+EA	194−204 216−226+EA 221−231+EA	0 2/4 4/6	0 6 8	FDIVR ST(2),ST FDIVR〔BX〕.PLUSE_RATE FDIVR RECORDER FREQUENCY	実数逆除算
203	198−208	0	0	FDIVRP ST(1),ST	実数逆除算とポップ
5	2 − 8	0	0	FNENI	インタラプト許可
11	9 −16	0	0	FFREE ST(1)	レジスタの解放
120+EA 125+EA	102−137+EA 108−143+EA	1/2 2/4	2 4	FIADD DISTANCE_TRAVELLED FIADD PLUSE_COUNT〔SI〕	整数加算
80+EA 85+EA	72−86+EA 78−91+EA	1/2 2/4	2 4	FICOM TOOL.N_PASSES FICOM〔BP+4〕.PARM_COUNT	整数比較
82+EA 87+EA	74−88+EA 80−93+EA	1/2 2/4	2 4	FICOMP〔BP〕.LIMIT〔SI〕 FICOMP N_SAMPLES	整数比較とポップ
230+EA 236+EA	224−238+EA 230−243+EA	1/2 2/4	2 4	FIDIV SURVEY. OBSERVATIONS FIDIV RELATIVE_ANGLE〔DI〕	整数除算

ニーモニック	オペランド	オペレーションコード																バイト数
		7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
FIDIVR	word-integer	ESCAPE					1	1	0	MOD		1	1	1	R/M			2−4
	short-integer	ESCAPE					0	1	0	MOD		1	1	1	R/M			2−4
FILD	word-integer	ESCAPE					1	1	1	MOD		0	0	0	R/M			2−4
	short-integer	ESCAPE					0	1	1	MOD		0	0	0	R/M			2−4
	long-integer	ESCAPE					1	1	1	MOD		1	0	1	R/M			2−4
FIMUL	word-integer	ESCAPE					1	1	0	MOD		0	0	1	R/M			2−4
	short-integer	ESCAPE					0	1	0	MOD		0	0	1	R/M			2−4
FINCSTP	−	ESCAPE					0	0	1	1	1	1	1	0	1	1	1	2
FINIT/ FNINIT	−	ESCAPE					0	1	1	1	1	1	0	0	0	1	1	2
FIST	word-integer	ESCAPE					1	1	1	MOD		0	1	0	R/M			2−4
	short-integer	ESCAPE					0	1	1	MOD		0	1	0	R/M			2−4
FISTP	word-integer	ESCAPE					1	1	1	MOD		0	1	1	R/M			2−4
	short-integer	ESCAPE					0	1	1	MOD		0	1	1	R/M			2−4
	long-integer	ESCAPE					1	1	1	MOD		1	1	1	R/M			2−4
FISUB	word-integer	ESCAPE					1	1	0	MOD		1	0	0	R/M			2−4
	short-integer	ESCAPE					0	1	0	MOD		1	0	0	R/M			2−4
FISUBR	word-integer	ESCAPE					1	1	0	MOD		1	0	1	R/M			2−4
	short-integer	ESCAPE					0	1	0	MOD		1	0	1	R/M			2−4
FLD	ST(i)	ESCAPE					0	0	1	1	1	0	0	0	ST(i)			2
	short-real	ESCAPE					0	0	1	MOD		0	0	0	R/M			2−4
	long-real	ESCAPE					1	1	1	MOD		1	0	1	R/M			2−4
	temp-real	ESCAPE					0	1	1	MOD		1	0	1	R/M			2−4
FLDCW	2-bytes	ESCAPE					0	0	1	MOD		1	0	1	R/M			2−4
FLDENV	14-bytes	ESCAPE					0	0	1	MOD		1	0	0	R/M			2−4
FLDLG2	−	ESCAPE					0	0	1	1	1	1	0	1	1	0	0	2
FLDLN2	−	ESCAPE					0	0	1	1	1	1	0	1	1	0	1	2
FLDL2E	−	ESCAPE					0	0	1	1	1	1	0	1	0	1	0	2
FLDL2T	−	ESCAPE					0	0	1	1	1	1	0	1	0	0	1	2
FLDPI	−	ESCAPE					0	0	1	1	1	1	0	1	0	1	1	2
FLDZ	−	ESCAPE					0	0	1	1	1	1	0	1	1	1	0	2
FLD1	−	ESCAPE					0	0	1	1	1	1	0	1	0	0	0	2
FMUL	// ST(i),ST/ST,ST(i)	ESCAPE					d	0	0	1	1	0	0	1	R/M			2
	// ST(i),ST/ST,ST(i)	ESCAPE					d	0	0	1	1	0	0	1	R/M			2
	short-real	ESCAPE					0	0	0	MOD		0	0	1	R/M			2−4
	long-real *	ESCAPE					1	0	0	MOD		0	0	1	R/M			2−4
	long-real	ESCAPE					1	0	0	MOD		0	0	1	R/M			2−4
FMULP	ST(i),ST *	ESCAPE					d	1	0	1	1	0	0	1	R/M			2
	ST(i),ST	ESCAPE					d	1	0	1	1	0	0	1	R/M			2
FNOP	−	ESCAPE					0	0	1	1	1	0	1	0	0	0	0	2
FPATAN	−	ESCAPE					0	0	1	1	1	1	1	0	0	1	1	2

クロック数		転送		コーディング例	コメント
Typical	Range	8086	8088		
230+EA 237+EA	225−239+EA 231−245+EA	1/2 2/4	2 4	FIDIVR〔BP〕.X_COORD FIDIVR FREQUENCY	整数逆除算
50+EA 56+EA 64+EA	46−54+EA 52−60+EA 60−68+EA	1/2 4/6 2/4	2 4 8	FILD〔BX〕.SEQUENCE FILD STANDOFF〔DI〕 FILD RESPONSE.COUNT	整数ロード
130+EA 136+EA	124−138+EA 130−144+EA	1/2 2/4	2 4	FIMUL BEARING FIMUL POSITION.Z_AXIS	整数乗算
9	6 −12	0	0	FINCSTP	スタックポインタのインクリメント
5	2 − 8	0	0	FINIT	NDP初期化
86+EA 88+EA	80−90+EA 82−92+EA	2/4 3/5	4 6	FIST OBS.COUNT〔SI〕 FIST〔BP〕.FACTORED_PULSES	整数ストア
88+EA 90+EA 100+EA	82−92+EA 84−94+EA 94−105+EA	2/4 3/5 5/7	4 6 10	FISTP〔BX〕.ALPHA_COUNT〔SI〕 FISTP CORRECTED_TIME FISTP PANEL.N_READINGS	整数ストアとポップ
120+EA 125+EA	102−137+EA 108−143+EA	1/2 2/4	2 4	FISUB BASE_FREQUENCY FISUB TRAIN_SIZE〔DI〕	整数減算
120+EA 125+EA	103−139+EA 109−144+EA	1/2 2/4	2 4	FISUBR FLOOR〔BX〕〔SI〕 FISUBR BALANCE	整数逆減算
20 43+EA 46+EA 57+EA	17−22 38−56+EA 40−60+EA 53−65+EA	0 2/4 4/6 5/7	0 4 8 10	FLD ST(0) FLD READING〔SI〕.PRESSURE FLD〔BP〕.TEMPERATURE FLD SAVEREADING	実数ロード
10+EA	7−14+EA	1/2	2	FLDCW CONTROL_WORD	コントロールワードのロード
40+EA	35−45+EA	7/9	14	FLDENV〔BP+6〕	エンバイアラメントのロード
21	18−24	0	0	FLDLG2	$\log_{10}2$のロード
20	17−23	0	0	FLDNL2	$\log_e 2$のロード
18	15−21	0	0	FLDL2E	$\log_2 e$のロード
19	16−22	0	0	FLDL2T	$\log_2 10$のロード
19	16−22	0	0	FLDPI	πのロード
14	11−17	0	0	FLDZ	0のロード
18	15−21	0	0	FLD1	1のロード
97 138 118+EA 120+EA 161+EA	90−105 130−145 110−125+EA 112−126+EA 154−168+EA	0 0 2/4 4/6 4/6	0 0 4 8 8	FMUL ST,ST(3) FMUL ST,ST(3) FMUL SPEED_FACTOR FMUL〔BP〕.HEIGHT FMUL〔BP〕.HEIGHT	実数乗算
100 142	94−108 134−148	0 0	0 0	FMULP ST(1),ST FMULP ST(1),ST	実数乗算とポップ
13	10−16	0	0	FNOP	ノーオペレーション
650	250−800	0	0	FPATAN	部分アークタンジェント

ニーモニック	オペランド	オペレーションコード																バイト数
		7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
FPREM	—	ESCAPE					0	0	1	1	1	1	1	1	0	0	0	2
FPTAN	—	ESCAPE					0	0	1	1	1	1	1	0	0	1	0	2
FRNDINT	—	ESCAPE					0	0	1	1	1	1	1	1	1	0	0	2
FRSTOR	94-bytes	ESCAPE					1	0	1	MOD		1	0	0	R/M			2−4
FSAVE/ FNSAVE	94-bytes	ESCAPE					1	0	1	MOD		1	1	0	R/M			2−4
FSCALE	—	ESCAPE					0	0	1	1	1	1	1	1	1	0	1	2
FSQRT	—	ESCAPE					0	0	1	1	1	1	1	1	0	1	0	2
FST	ST(i)	ESCAPE					1	0	1	1	1	0	1	0	ST(i)			2
	short-real	ESCAPE					0	0	1	MOD		0	1	0	R/M			2−4
	long-real	ESCAPE					1	0	1	MOD		0	1	0	R/M			2−4
FSTCW/ FNSTCW	2-bytes	ESCAPE					0	0	1	MOD		1	1	1	R/M			2−4
FSTENV/ FNSTENV	14-bytes	ESCAPE					0	0	1	MOD		1	1	0	R/M			2−4
FSTP	ST(i)	ESCAPE					1	0	1	1	1	0	1	1	ST(i)			2
	short-real	ESCAPE					0	0	1	MOD		0	1	1	R/M			2−4
	long-real	ESCAPE					1	0	1	MOD		0	1	1	R/M			2−4
	temp-real	ESCAPE					0	1	1	MOD		1	1	1	R/M			2−4
FSTSW/ FNSTSW	2-bytes	ESCAPE					1	0	1	MOD		1	1	1	R/M			2−4
FSUB	// ST,ST(i)/ST(i),ST	ESCAPE					d	0	0	1	1	1	0	0	R/M			2
	short-real	ESCAPE					0	0	0	MOD		1	0	0	R/M			2−4
	long-real	ESCAPE					1	0	0	MOD		1	0	0	R/M			2−4
FSUBP	ST(i),ST	ESCAPE					d	1	0	1	1	1	0	0	R/M			2
FSUBR	// ST,ST(i)/ST(i),ST	ESCAPE					d	0	0	1	1	1	0	1	R/M			2
	short-real	ESCAPE					0	0	0	MOD		1	0	1	R/M			2−4
	long-real	ESCAPE					1	0	0	MOD		1	0	1	R/M			2−4
FSUBRP	ST(i),ST	ESCAPE					d	1	0	1	1	1	0	1	R/M			2
FTST	—	ESCAPE					0	0	1	1	1	1	0	0	1	0	0	2
FWAIT	—	1	0	0	1	1	0	1	1	—								1
FXAM	—	ESCAPE					0	0	1	1	1	1	0	0	1	0	1	2
FXCH	// ST(i)	ESCAPE					0	0	1	1	1	0	0	1	ST(i)			2
FXTRACT	—	ESCAPE					0	0	1	1	1	1	1	0	1	0	0	2
FYL2X	—	ESCAPE					0	0	1	1	1	1	1	0	0	0	1	2
FYL2XP1	—	ESCAPE					0	0	1	1	1	1	1	1	0	0	1	2
F2XM1	—	ESCAPE					0	0	1	1	1	1	1	0	0	0	0	2

注） d(Destination) = { 0 … Destination is ST(0) / 1 … // is ST(i) }
＊オペランド一方あるいは両方が「short」のとき
▲nは8087がbusyであることをCPUが調べた回数

クロック数		転送		コーディング例	コメント
Typical	Range	8086	8088		
125	15－190	0	0	FPREM	部分剰余
450	30－540	0	0	FPTAN	部分タンジェント
45	16－50	0	0		整数化
210＋EA	205－215＋EA	47/49	96	FRSTOR[BP]	ステートの回復
210＋EA	205－215＋EA	48/50	94	FSAVE[BP]	ステートのセーブ
35	32－38	0	0	FSCALE	スケール (2のべき乗倍)
183	180－186	0	0	FSQRT	平方根
18 87＋EA 100＋EA	15－22 84－90＋EA 96－104＋EA	0 3/5 5/7	0 6 10	FST ST(3) FST CORRELATION[DI] FST MEAN_READING	実数ストア
15＋EA	12－18＋EA	2/4	4	FSTCW SAVE_CONTROL	コントロール ワードのストア
45＋EA	40－50＋EA	8/10	16	FSTENV[BP]	エンバイアラメン トのストア
20 89＋EA 102＋EA 55＋EA	17－24 86－92＋EA 98－106＋EA 52－58＋EA	0 3/5 5/7 6/8	0 6 10 12	FSTP ST(2) FSTP[BX].ADJUSTED_RPM FSTP TOTAL_DOSAGE FSTP REG_SAVE[SI]	実数ストアと ポップ
15＋EA	12－18＋EA	2/4	4	FSTSW SAVE_STATUS	ステータスワード のストア
85 105＋EA 110＋EA	70－100 90－120＋EA 95－125＋EA	0 2/4 4/6	0 4 8	FSUB ST,ST(2) FSUB BASE_VALUE FSUB COORDINATEX	実数減算
90	75－105	0	0	FSUBP ST(2),ST	実数減算とポップ
87 105＋EA 110＋EA	70－100 90－120＋EA 95－125＋EA	0 2/4 4/6	0 4 8	FSUBR ST,ST(1) FSUBR VECTOR[SI] FSUBR[BX]. INDEX	実算逆減算
90	75－105	0	0	FSUBRP ST(1),ST	実数逆減算と ポップ
42	38－48	0	0	FTST	0との比較
3＋5n▲	3＋5n▲	0	0	FWAIT	8087の 動作終了待ち
17	12－23	0	0	FXAM	スタックトップ の調査
12	10－15	0	0	FXCH ST(2)	レジスタの交換
50	27－55	0	0	FXTRACT	指数部と 有効数字の分離
950	900－1100	0	0	FYL2X	$\log_2 X$
850	700－1000	0	0	FYL2XP1	$\log_2(X+1)$
500	310－630	0	0	F2XM1	2^X-1

メモリアドレシング

mod / r/m	0 0	0 1	1 0
0 0 0	(BX)+(SI)	(BX)+(SI)+disp8	(BX)+(SI)+disp16
0 0 1	(BX)+(DI)	(BX)+(DI)+disp8	(BX)+(DI)+disp16
0 1 0	(BP)+(SI)	(BP)+(SI)+disp8	(BP)+(SI)+disp16
0 1 1	(BP)+(DI)	(BP)+(DI)+disp8	(BP)+(DI)+disp16
1 0 0	(SI)	(SI)+disp8	(SI)+disp16
1 0 1	(DI)	(DI)+disp8	(DI)+disp16
1 1 0	DIRECT ADDRESS	(BP)+disp8	(BP)+disp16
1 1 1	(BX)	(BX)+disp8	(BX)+disp16

フラグ動作の略称

識 別 子	説 明
(ブランク)	変化なし
0	0にクリアされる
1	1にセットされる
X	結果に従ってセットまたはクリアされる
U	不 定
R	以前に退避した値がリストアされる

セグメントレジスタの選択

sreg	
0 0	E S
0 1	C S
1 0	S S
1 1	D S

8/16ビット汎用レジスタの選択

reg or r/m *	W = 0	W = 1
0 0 0	A L	A X
0 0 1	C L	C X
0 1 0	D L	D X
0 1 1	B L	B X
1 0 0	A H	S P
1 0 1	C H	B P
1 1 0	D H	S I
1 1 1	B H	D I

*r/mはmodのない場合

8086アセンブリ言語

―8086 ASSEMBLY LANGUAGE―

初版印刷　昭和61年1月15日
初版発行　昭和61年1月18日

著　　者　西村義孝

発 行 者　孫正義
発 行 所　株式会社日本ソフトバンク
出　版　部
東京都千代田区四番町2－1（〒102）
電話：03（261）4095
定　　価　カバーに記載されております
印 刷 所　奥村印刷株式会社

（ISBN 4-930795-27-2）

NEC
PCシリーズ関連書籍

DISK CHARGE

100%ディスク
活用の手引き書

木下淳博 著

定価1,800円(〒250)

B5判・216ページ

ディスク利用上，データファイルを扱う技術はプログラミング技術以上のものを要求されます。
本書はファイルの基礎からデータ構造，ソーティング，データ圧縮に至るまで豊富な内容を満載しています。サンプルプログラムはPC用ですが，少しの変更でシャープX1や富士通FM7などにも使えます。ディスク利用者に必携の書となるでしょう。

ALL
DISK
USER